银领工程

高等职业教育技能型紧缺人才培养培训工程系列教材

软件编程规范

徐人凤　孙宏伟　王　梅

KD0025385

高等教育出版社

“十五”国家级规划教材、教育部高职高专规划教材

高等职业教育技能型紧缺人才培养培训系列教材

计算机公共基础教程（第二版)(赠电子教案）	李存斌
计算机公共基础教程上机实验指导（第二版)(配盘）	李存斌
计算机公共基础—基本知识和使用	高　林
计算机公共基础—Word 2000	高　林
计算机公共基础—Excel 2000	高　林
计算机公共基础—PowerPoint 2000	高　林
计算机公共基础—WPS 2000	高　林
计算机应用基础（赠电子教案）	宋清龙
计算机公共基础（赠电子教案）	刘　钢
计算机公共基础实训指导（配盘）	刘　钢
计算机基础应用(2003 版)(配盘 E-Textbook)	黄旭明
IT 职业素养	雷　瑛
计算机英语(配盘 E-Textbook)	邱仲潘
计算机数学基础(配盘 E-Textbook)	叶东毅
数字电路与逻辑设计（第二版)(赠电子教案）	胡　锦
微机原理及其应用（第二版）	丁新民
微型计算机原理（第二版)(赠电子教案）	宋汉珍
计算机组成原理及汇编语言	张思发
计算机组成原理及汇编语言学习指导（配盘）	张思发
汇编语言程序设计	周学毛
微机接口技术（第二版)(赠电子教案）	王成端
数据结构（第二版）(配盘）	陈　雁
数据结构（配套网络课程）(配盘）	蒋文蓉

计算机维护与维修（第二版)(赠电子教案）	曹　哲
计算机与信息技术概论	曹晓川
计算机硬件技术基础	杨根兴
程序设计基础——逻辑编程及 C++ 实现	陆　虹
程序设计基础——逻辑编程及 C++ 实现实训教程	陆　虹
程序设计基础——面向对象及 C++ 实现	贾振华
程序设计基础——可视化及 VC++ 实现	周晓云
VB 程序设计	沈祥玖
VB 程序设计及应用（配盘）	李淑华
PowerBuilder 程序设计及应用	孙秋冬
Delphi 程序设计（配盘）	周志德
Java 编程及应用	杨　武
Java 面向对象程序设计	聂　哲
ASP 编程技术基础（配盘）	李存斌
JSP 程序设计（配盘）	蒋文蓉
网络技术基础—— Internet 与网页设计	童　欣
多媒体技术及应用	袁小红
操作系统—— Windows 2000	方　程
多用户操作系统—— Windows 2000 Server	傅连仲
数据库基础——基于 MS Access 的数据库设计(配盘)	沈祥玖
SQL Server 2000 数据库及应用（配盘）	徐人凤
Oracle 大型数据库及应用	李卓玲
软件测试	赵瑞莲
软件工程概论	陶华亭
软件技术基础	来可伟
软件编程规范	徐人凤
UML 技术及应用	丁　峰

ISBN 7-04-016990-8

9 787040 169904 >

定价 17.90 元

高等职业教育技能型紧缺人才培养培训工程系列教材

软件编程规范

徐人凤　孙宏伟　王　梅

高等教育出版社

内容提要

本书是高等职业教育技能型紧缺人才培养培训工程系列教材，是编者在总结多年从事企业软件项目开发经验和目前一些大的软件企业的软件编程规范的基础上编写而成的。

本书内容全面、实用，按 C、C++ 、VC++、Java、Delphi 等语言分类编写，给出这些语言中关于变量命名、注释、函数/过程的书写、错误和异常处理等编写规范。本书注重培养编程人员的实际应用能力，以实例贯穿整个章节。同时，提供了很多有关编程规范方面的正例与反例，让读者对比、了解，从而提高自己的编程技能，并逐步养成良好的编程习惯。

本书可作为普通高校、高等职业学校、高等专科学校、成人高校、本科院校举办的二级职业技术学院等所有涉及软件编程课程的学生的教材和参考书，也可供示范性软件职业技术学院、继续教育学院、民办高校、技能型紧缺人才培养使用。另外，还能作为软件编程人员的参考资料和软件公司的培训教材。

图书在版编目(CIP)数据

软件编程规范/徐人凤，孙宏伟，王梅. —北京：
高等教育出版社，2005.7

ISBN 7-04-016990-8

Ⅰ.软... Ⅱ.①徐...②孙...③王... Ⅲ.软件设计-规范-高等学校：技术学校-教材
Ⅳ.TP311.5-65

中国版本图书馆 CIP 数据核字(2005) 第 056000 号

策划编辑	冯 英	责任编辑	张海波	封面设计	王凌波
版式设计	胡志萍	责任校对	殷 然	责任印制	韩 刚

出版发行	高等教育出版社	购书热线	010-58581118
社 址	北京市西城区德外大街 4 号	免费咨询	800-810-0598
邮政编码	100011	网 址	http://www.hep.edu.cn
总 机	010-58581000		http://www.hep.com.cn
经 销	北京蓝色畅想图书发行有限公司	网上订购	http://www.landraco.com
印 刷	廊坊市文峰档案文化用品有限公司		http://www.landraco.com.cn
开 本	787×1092 1/16	版 次	2005 年 7 月第 1 版
印 张	14	印 次	2005 年 7 月第 1 次印刷
字 数	330 000	定 价	17.90 元

本书如有缺页、倒页、脱页等质量问题，请到所购图书销售部门联系调换。

版权所有 侵权必究

物料号 16990-00

出 版 说 明

为了认真贯彻《国务院关于大力推进职业教育改革与发展的决定》,落实《2003—2007 年教育振兴行动计划》,缓解国内劳动力市场技能型人才紧缺现状,为我国走新型工业化道路服务,自 2001 年 10 月以来,教育部在永州、武汉和无锡连续三次召开全国高等职业教育产学研经验交流会,明确了高等职业教育要“以服务为宗旨,以就业为导向,走产学研结合的发展道路”,同时明确了高等职业教育的主要任务是培养高技能人才。这类人才,既要能动脑,更要能动手,他们既不是白领,也不是蓝领,而是应用型白领,是“银领”。从而为我国高等职业教育的进一步发展指明了方向。

培养目标的变化直接带来了高等职业教育办学宗旨、教学内容与课程体系、教学方法与手段、教学管理等诸多方面的改变。与之相应,也产生了若干值得关注与研究的新课题。对此,我们组织有关高等职业院校进行了多次探讨,并从中遴选出一些较为成熟的成果,组织编写了“银领工程”丛书。本丛书围绕培养符合社会主义市场经济和全面建设小康社会发展要求的“银领”人才的这一宗旨,结合最新的教改成果,反映了最新的职业教育工作思路和发展方向,有益于固化并更好地推广这些经验和成果,很值得广大高等职业院校借鉴。我们的这一想法和做法也得到了教育部领导的肯定,教育部副部长吴启迪专门为首批“银领工程”丛书提笔作序。

我社出版的高等职业教育各专业领域技能型紧缺人才培养培训工程系列教材也将陆续纳入“银领工程”丛书系列。

“银领工程”丛书适用于高等职业学校、高等专科学校、成人高校及本科院校举办的二级职业技术学院、继续教育学院和民办高校使用。

高等教育出版社

2004 年 9 月

自　序

软件产业是我国重点支柱产业之一，软件生产规范化、标准化同国际软件生产接轨是软件产业迅速扩大生产规模的必经之路。

印度之所以能建成拥有上万名员工的“软件工厂”，将软件作为产品，从软件生产线上生产出来，成为软件出口大国，一个非常重要的原因就是，印度的很多软件企业将质量认证作为进军国际软件市场的通行证，其软件编程实现了规范化。

我国许多软件工程师往往片面追求软件编程中算法结构的技巧，或按照自己的编程习惯书写程序代码，如程序格局、变量名的定义、大、小写字母的使用等随自己的喜欢而定。这都与国外软件行业认可的软件编程标准相违背，其结果必然造成编写出的程序代码可读性差，其产品也很难向国际市场推广。因为从规范化角度讲，这样的程序代码本身就不是产品。另外，研制一个大型的软件系统时，团队成员按照各自习惯编写出来的程序代码，在进行系统集成时，会出现很多问题，甚至使整个系统的开发前功尽弃。

看一看用微软、Borland 等公司工具自动生产的代码和具有丰富经验的程序员编写的程序，你就会发现，它确实让人感觉很“舒服”。这是为什么呢？

原因其实很简单，那就是这些程序代码遵循相同的软件编程规范，具有统一的编程风格。只有按照一定的规范进行编程，才能使编写出的程序代码能让其他人读得懂，也只有这样的程序代码才有可能进行系统集成。

我国软件产品要想走向世界，必须要遵循软件编程规范。目前，国内一些大的软件公司已经意识到这个问题，并且在公司内部建立起自己的软件编程规范。而一些中、小型软件公司即使意识到这个问题，也由于多种原因的制约无法在短期内实现软件编程的规范化。

编写本软件编程规范的目的，是为了使程序员具有良好的程序代码编写风格，将规范作为企业编程中要遵守的“法规”，以保证软件产品的高质量，提高开发效率。使用软件编程规范，可以使团队遵守相同的编程规则、具有统一的编程风格，多个人写出的程序代码看上去就像是同一个程序员编写的一样。代码易于为他人所理解，从而会在很大程度上提高代码的可维护性，也降低了以后的维护成本。

软件的生命周期中有 80% 的时间用于维护，而且维护者往往不是代码编写者本人。遵循软件编程规范编程可以提高代码的可读性，方便其他人员理解和读懂，从而在很大程度上提高代码的可维护性、降低维护成本。另外，在开发系列软件产品时，软件编程规范的应用将能保证所有的软件产品具有一致的风格。

软件编程规范的使用简化了程序员的工作，“简化”的含义不是减少代码量（相反，很多时候遵从规范会带来更多的代码），而是减少程序员在维护代码时的劳动量。另外，使用规范在很大程度上也是为了减少程序员的记忆负担。

前　言

目前，在国内各院校的软件人才培养中，涉及“如何设计和编写高质量的程序”的课程并不多，市场上相关书籍也十分匮乏。因此，教师在教学过程中以及学生在学习和编程实践过程中都很少能自觉地关注软件的质量和代码的风格。一些工作在中、小型企业的程序员，只有靠勤奋好学，并有意识地总结程序开发中的经验和教训，才能积累一些有关规范化编程的注意事项。

编写本书的目的就是为在校学生和初入软件行业的新手提供帮助，使他们掌握软件编程规范，养成良好的编程风格，以编写出高质量的程序。

本书由一条条的规则组成，基本上每条规则都是相对独立的，所以可以按任何顺序阅读或在需要时查阅，以掌握软件编程规范。

本书讲述 C、C++ 、VC++ 、Delphi、Java 等 5 种语言的软件编程规范，共八百多条规范，它由 4 个部分构成。

第一部分为 C/C++ 语言编程规范。该部分由 11 章组成。主要内容包括：程序约定；变量、常量与数据类型；表达式和基本语句；函数和过程；内存和指针；类和类函数；类的继承；可测试性；程序效率和质量保证；错误和异常处理规范以及其他规范。

第二部分为 Java 语言编程规范。该部分由 8 章组成。主要内容包括：程序组织规则；命名约定；注释约定；变量、常量；表达式和基本语句；类和类方法；代码规范以及其他规范。

第三部分为 Delphi 语言编程规范。该部分由 7 章组成。主要内容包括：程序约定；变量、常量与类型；语句；函数和过程；类和类方法；界面设计及其他规范。

第四部分为 VC++ 编程规范。该部分由 4 章组成。内容包括：编程环境设置；布局及变量；头文件、注释及其他；优化编码等。

本书附录的主要内容为 C/C++ 规范检查表、Delphi 标准控件的命名参考及 VC++ 优化编码实现过程示例。

规范并不是永远不变的，可以按照“易于使用”的原则来修改。许多大公司在某些细节上都制定了自己的规则。同时，程序设计工具对编程规范也有很大的影响，这个影响来源于开发商的程序设计风格。同样基于 C++ ，在 Microsoft Visual C++ 和 Borland C++ Builder 中就使用了不完全相同的编程规范。Microsoft 和 Borland 有着各自不同的、而且十分鲜明的风格。作为用户，我们可以在此基础上有所改变，但是这是有限度的。其实，在用户做出对供应商和开发工具的选择时，同时也就确定了其开发程序的风格。

另外需要强调的是，本书源于有关程序开发组织的约定及许多程序员的经验，所以本书试图为读者提供一个参考，希望读者在理解每一条规则的基础上，在自己的工作实践中加以灵活运用。

关于本书的使用方式，建议有以下三种方法：

第一种，作为软件技术专业、网络技术专业、计算机应用等专业学生的必读教材。在学习 C 语言程序设计、C++ 、Delphi、Java 时，可随时学习和查阅。

第二种，可作为有关选修课程的教材，建议学时为32学时。

第三种，也可作为软件企业的编程人员的工具书，在学习和编程实践中随时翻阅。

在本书编写的过程中，得到许多人的热情支持和帮助。

在此非常感谢聂哲先生和袁梅冷女士，他们两位对Java语言精髓的深入理解和编程经验，对本书Java语言部分进行了认真的审核，提出了许多宝贵的建议。

同时也要感谢李俊平和曾建华两位先生，他们多年来对Delphi语言的使用积累了非常丰富的软件开发经验，并提供给本书，无私地与大家分享。

在这里，也要感谢钟剑龙先生，他为本书提供了部分素材。

最后，非常感谢李俊平先生，他在繁忙的工作中抽出时间审阅了全稿。

真诚感谢在本书编写过程中，给予我们帮助和支持的家人和朋友，感谢他们的理解和配合。

本书由深圳职业技术学院徐人凤（高级工程师）、孙宏伟（工程师）、王梅（讲师）共同编写，由徐人凤对全书进行了统稿。徐人凤和孙宏伟均具有多年在企业从事软件开发的实际经验。

由于时间仓促和编者水平有限，书中难免有错漏之处，敬请读者批评指正，在此表示诚挚的谢意。编者 E－Mail：xurf@oa.szpt.net。

编　者

2005年3月于深圳

目　录

第一部分　C/C++ 语言编程规范

第二部分　Java 语言编程规范

第三部分　Delphi 语言编程规范

第四部分　VC++ 编程规范

第一部分

C/C++ 语言编程规范

第1章　程序约定

通常，一个 C/C++ 程序分为两个文件。一个文件用于保存程序的声明(declaration)代码，称为头文件。另一个文件用于保存程序的实现(implementation)代码，称为定义文件。

C/C++ 程序的头文件以“.H”为后缀，C 程序的定义文件以“.C”为后缀，C++ 程序的定义文件通常以“.CPP”为后缀(也有一些系统以“.CC”或“.CXX”为后缀)。

1.1　程序排版布局规则

程序排版布局的目的是显示出程序良好的逻辑结构，提高程序的准确性、连续性、可读性和可维护性。更重要的是，统一的程序布局和编程风格，有助于提高整个软件的开发质量，提高开发效率，降低开发成本。同时，对于普通程序员来说，养成良好的编程习惯有助于提高自己的编程水平和编程效率。

【规则 1-1-1】程序块应采用缩进格式编写，缩进的空格数为 4 个。

说明　对于由开发工具自动生成的代码可不遵循此规则。

【规则 1-1-2】程序的分界符“{”和“}”应独占一行并且位于同一列，同时与引用它们的语句左对齐。{ }之内的代码块使用缩进规则对齐。

说明　这样能使代码便于阅读，并且方便注释。但 do while 语句和结构的类型化时可以例外，while 条件和结构名可与“}”在同一行。

反例

```
void Func (int iVar){
    while (condition){
        DoSomething( );
    }}
```

正例

```
void Func (int iVar)
{                                   // 独占一行并与引用语句左对齐
    while (condition)
    {
        DoSomething( );     // 以{、}为准缩进 4 格
    }
}
```

【规则1-1-3】一行代码只完成一个功能,即一个变量定义占一行,一个语句占一行。

反例

```
double width, height, depth;          // 定义宽度、高度及深度变量
```

正例

```
double width;      // 定义宽度变量
double height;     // 定义高度变量
double depth;      // 定义深度变量
```

【规则1-1-4】相对独立的程序块之间、变量说明之后必须加空行。

反例

```
if (a > b)
{
        … // program code
            max = a;
}
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

正例

```
if (!valid_ni(ni))
{
        … // program code
}

repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

【规则1-1-5】程序中一行代码及其注释的总字符数不能超过80列。

说明 包括空格在内的该行总字符数不超过80列。

【规则1-1-6】较长的语句(超过80字符)要分成多行书写,长表达式应在低优先级操作符处拆分新行,操作符放在新行之首,划分出的新行要进行适当的缩进,使排版整齐、语句可读。

说明 ① 条件表达式的续行在第一个条件处对齐。

② for循环语句的续行在初始化条件语句处对齐。

③ 函数调用和函数声明的续行在第一个参数处对齐。

④ 赋值语句的续行应在赋值号处对齐。

正例 1

```
perm_count_msg. head. len = NO7_TO_STAT_PERM_COUNT_LEN
                          + STAT_SIZE_PER_FRAM * sizeof( _UL );

act_task_table[ frame_id * STAT_TASK_CHECK_NUMBER + index]. occupied
                = stat_poi[ index]. occupied;

act_task_table[ taskno]. duration_true_or_false
                = SYS_get_sccp_statistic_state( stat_item );

report_or_not_flag = ((taskno < MAX_ACT_TASK_NUMBER)
                    && (n7stat_stat_item_valid (stat_item))
                    && (act_task_table[ taskno]. result_data ! = 0));
```

正例 2

```
if ((iFormat == CH_A_Format_M)
    && (iOfficeType == CH_BSC_M)) // 条件表达式的续行在第一个条件处对齐
{
    DoSomething();
}

for (long_initialization_statement;
     long_condiction_statement;          // for 循环语句续行在初始化条件语句处对齐
     long_update_statement)
{
    DoSomething();
}

// 函数声明的续行在第一个参数处对齐
BYTE ReportStatusCheckPara(HWND hWnd,
                           BYTE ucCallNo,
                           BYTE ucStatusReportNo);

// 赋值语句的续行应在赋值号处对齐
fTotalBill = fTotalBill + faCustomerPurchases[ iID]
           + fSalesTax(faCustomerPurchases[ iID]);
```

【规则 1 -1 -7】循环、判断等语句中若有较长的表达式或语句，则要进行适当的拆分，长表达式应在低优先级操作符处拆分成新行，操作符放在新行之首。

正例

```
if ((taskno < max_act_task_number)
        && (n7stat_stat_item_valid (stat_item)))
{
        … // program code
}
```

```
for (i =0, j = 0; (i <BufferKeyword[ word_index ]. word_length)
                        && (j < NewKeyword. word_length); i++ , j++ )
{
        … // program code
}

for (i = 0, j = 0;
        (i < first_word_length) && (j < second_word_length);
        i++ , j++ )
{
        … // program code
}
```

【规则1-1-8】定义指针类型的变量时,应使“*”紧挨着变量名。对“&”也采取同样的处理规则。

反例

```
float *    pfBuffer;
```

正例

```
float    * pfBuffer;
int    * x, y; // 此处 y 不会被误解为指针
```

【规则1-1-9】若函数或过程中的参数较长,则要进行适当的拆分。

正例

```
n7stat_str_compare( (BYTE * ) & stat_object,
                        (BYTE * ) & (act_task_table[ taskno ]. stat_object),
                        sizeof (_STAT_OBJECT));

n7stat_flash_act_duration( stat_item, frame_id * STAT_TASK_CHECK_NUMBER
+ index, stat_object );
```

【规则1-1-10】不允许把多个短语句写在一行中,即一行只写一条语句。

反例

```
rect. length = 0; rect. width = 0;
```

正例

```
rect. length = 0;
rect. width = 0;
```

【规则 1-1-11】if、else、else if、for、while、do、case、switch、default 等语句独占一行，执行语句不得紧跟其后。不论执行语句有多少条都要加{}。

说明　这样可以防止书写错误，也易于阅读。

反例　下面的代码执行语句紧跟 if 的条件之后，而且没有加{}，违反规则。

```
if (width < height) dosomething();
```

正例

```
if (width < height)
{
dosomething();
}
```

【规则 1-1-12】源程序中关系较为紧密的代码应尽可能相邻。

说明　这样便于程序阅读和查找。

反例

```
iLength = 10;
strCaption = "Test";
iWidth = 5;
```

正例

```
iLength = 10;
iWidth = 5;      // 矩形的长与宽关系较密切，放在一起
StrCaption = "Test";
```

【规则 1-1-13】尽可能在定义变量的同时初始化该变量。

说明　如果变量的引用处和其定义处相隔比较远，变量的初始化工作很容易被忘记。如果引用了未被初始化的变量，可能导致程序错误。

正例

```
int width = 10; // 定义并初始化 width
int height = 10; // 定义并初始化 height
int depth = 10; // 定义并初始化 depth
```

【规则 1-1-14】禁止使用 Tab 键，必须使用空格键进行缩进。缩进量为 4 个空格。

说明　在使用不同的编辑器阅读程序时，应避免使用 Tab 键来设置空格，以免因空格数目的

不同而造成程序布局的不整齐。不要使用 BC 作为编辑器，因为 BC 会自动将 8 个空格变为一个 Tab 键，因此使用 BC 编辑器时会使程序缩进变乱。有的代码编辑器可以设置用空格代替 Tab 键。

【规则 1-1-15】函数或过程的开始、结构的定义及循环、判断等语句中的代码都要采用缩进格式。在 switch 语句中，每一个 case 分支和 default 都应用{ }括起来，而且{ }中的内容也需要缩进。

说明　这样做的目的是使程序可读性更好。

正例

```
switch (iCode)
    {
        case 1:
            {
                DoSomething();      // 缩进 4 格
                break;
            }
        case 2:
            {                               // 每一个 case 分支和 default 要用{ }括起来
                DoOtherThing();
                break;
            }
        ...                                 // 其他 case 分支
        default:
            {
                DoNothing();
                break;
            }
    }
```

【规则 1-1-16】声明类的时候，public、protected、private 等关键字应与分界符“{”、“}”对齐，这些部分的内容要进行缩进处理。

正例

```
class CCount
{
public:                                 // 与{ 对齐
    CCount (void);                      // 缩进处理
    ~CCount (void);
    int GetCount(void);
    void SetCount(int iCount);
```

```
    private:
        int m_iCount;
    }
```

【规则 1-1-17】在对两个以上的关键字、变量、常量进行对等操作时，它们之间的操作符之前、之后或者前后都要加空格；进行非对等操作时，如果是关系密切的立即操作符（如->），后面不应加空格。

说明　采用这种松散方式编写代码的目的是使代码更加清晰。由于留空格所产生的清晰性是相对的，所以，在已经非常清晰的语句中没有必要再留空格。如果语句已足够清晰，则括号内侧（即左括号后面和右括号前面）不需要加空格，多重括号间不必加空格，因为在 C/C++ 语言中括号已经是最清晰的标志了。在长语句中，如果需要加的空格非常多，那么应该保持整体清晰，而在局部不加空格。给操作符留空格时不要连续留两个以上的空格。

正例　① 只能在逗号、分号的后面加空格。

```
int a, b, c;
```

② 在比较操作符、赋值操作符（“=”、“+=”）、算术操作符（“+”、“%”）、逻辑操作符（“&&”、“&”）及位域操作符（“<<”、“^”）等双目操作符的前后都应加空格。

```
if (current_time >= MAX_TIME_VALUE)
a = b + c;
a *= 2;
a = b ^ 2;
```

③ 应在 if、for、while、switch 等关键字与后面的括号间加空格，这样可使关键字更为突出、明显。

```
if (a >= b && c > d)
```

④ 应在 const、virtual、inline、case 等关键字之后至少留一个空格，否则无法辨认这些关键字。

【规则 1-1-18】对于结构型数组及多维数组，如果在定义时进行初始化，则应按照数组的矩阵结构分行书写。

正例

```
int aiNumbers[4][3] =
{
        1, 1, 1,
        2, 4, 8,
        3, 9, 27,
        4, 16, 64
}
```

【规则 1-1-19】相关的赋值语句等号应对齐。

正例

```
tPDBRes.wHead    =  0;
```

```
tPDBRes. wTail      =   wMaxNumOfPDB - 1;
tPDBRes. wFree      =   wMaxNumOfPDB;
tPDBRes. wAddress   =   wPDBAddr;
tPDBRes. wSize      =   wPDBSize;
```

【规则 1 -1 -20】在每个类声明之后、每个函数实现结束之后都要加空行。

说明 空行起着分隔程序段落的作用,适当的空行可以使程序的布局更加清晰。

反例

```
void Foo::Hey(void)
{
    //Hey 实现代码
}
void Foo::Ack(void)
{
    //Ack 实现代码
}
// 两个函数的实现是两个逻辑程序块,应该用空行加以分隔。
```

正例

```
void Foo::Hey(void)
{
    //Hey 实现代码
}
        // 空一行
void Foo::Ack(void)
{
    //Ack 实现代码
}
```

【规则 1 -1 -21】在一个函数体内,逻辑上密切相关的语句之间不加空行,其他语句间应加空行分隔。

正例

```
// 空行
while (condition)
{
statement1;
// 空行
if (condition)
{
statement2;
```

```
    }
    else
    {
    statement3;
    }
    // 空行
    statement4;
}
```

【规则1-1-22】一元操作符,如“!”、“~”、“++”、“--”、“*”、“&”(地址运算符)等操作符前后不加空格。“[]”、“.”、“->”等操作符前后均不加空格。

正例

```
!bValue
~iValue
++iCount
*strSource
&fSum
aiNumber[i]=5;
tBox.dWidth
tBox->dWidth
```

【规则1-1-23】多元运算符(如“=”、“+=”、“>=”、“<=”、“+”、“*”、“%”、“&&”、“||”、“<<”,“^”等)和它们的操作数之间至少需要一个空格。

正例

```
fValue = fOldValue;
fTotal + fValue
iNumber += 2;
```

【规则1-1-24】函数名之后不要留空格。

说明　函数名后紧跟左括号“(”,以与关键字区别。

【规则1-1-25】“(”向后紧跟,“)”、“,”、“;”向前紧跟,紧跟处不留空格。“,”之后要留空格。“;”不是行结束符号时其后要留空格。

正例　下面例子中的“␣”代表空格。

```
for␣(i␣=␣0;␣i␣<␣MAX_BSC_NUM;␣i++)
{
    DoSomething(iWidth,␣iHeight);
}
```

【规则 1 -1 -26】注释符与注释内容之间要用一个空格进行分隔。

反例

```
/*注释内容*/
//注释内容
temp = (a > b)? a: b;          /*将 a 和 b 中的大者存入 temp 中*/
max = (temp > c)? temp: c;     //将 a 和 b 中的大者与 c 比较，取最大者存入 max 中
```

正例

```
/* 注释内容 */
// 注释内容

temp = (a > b)? a: b;          /* 将 a 和 b 中的大者存入 temp 中 */
max = (temp > c)? temp: c;     // 将 a 和 b 中的大者与 c 比较，取最大者存入 max 中
```

【规则 1 -1 -27】函数声明时，类型与名称不允许分行书写。

反例

```
extern double FAR
    CalcArea(double dWidth, double dHeight);
```

正例

```
extern double FAR CalcArea(double dWidth, double dHeight);
```

【规则 1 -1 -28】遵循统一的布局顺序来书写头文件及实现文件。

说明　以下内容如果某些节不需要，可以忽略。但是其他节要保持该次序。

① 头文件布局：

- 文件头(参见规范 1 -3 -4)
- #ifndef 文件名_H(全大写)
- #define 文件名_H
- 其他条件编译选项
- #include(依次为标准库头文件、非标准库头文件)
- 常量定义
- 全局宏
- 全局数据类型
- 类定义
- 模板(template)(包括 C++ 中的类模板和函数模板)
- 全局函数原型
- #endif

② 实现文件布局：

- 文件头
- #include(依次为标准库头文件、非标准库头文件)
- 常量定义
- 文件内部使用的宏
- 文件内部使用的数据类型
- 全局变量
- 本地变量(即静态全局变量)
- 局部函数原型
- 类的实现
- 全局函数
- 局部函数

【规则1-1-29】使用注释块来分隔头文件及实现文件中的各部分内容。

正例

```
/*************************************************************************
*                              数据类型定义                               *
*************************************************************************/
typedef unsigned char BOOLEAN;

/*************************************************************************
*                              函数原型                                   *
*************************************************************************/
int DoSomething(void);
```

【规则1-1-30】头文件必须避免重复包含。

说明　可以通过宏定义来避免重复包含。

正例

```
#ifndef MODULE_H
#define MODULE_H

//文件体

#endif
```

【规则1-1-31】包含标准库头文件时应用尖括号(< >),包含非标准库头文件时应用双引号(" "),以节省查找时间。

说明　尖括号和双引号的区别是:用尖括号时,系统到存放C/C++库函数头文件所在的目录中寻找要包含的文件,这种查找方式称为标准方式;用双引号时,系统先在用户当前目录中寻

找要包含的文件,若找不到,再按标准方式查找。自定义的头文件一般都在自己的工作目录下,用尖括号将导致查找失败。

正例

```
#include  < stdio. h >
#include " head. h"
```

【规则 1 -1 -32】遵循统一的顺序书写类的定义及实现。

说明　① 类的定义(在定义文件中)按如下顺序书写:

- 公有属性
- 公有函数
- 保护属性
- 保护函数
- 私有属性
- 私有函数

② 类的实现(在实现文件中)按如下顺序书写:

- 构造函数
- 析构函数
- 公有函数
- 保护函数
- 私有函数

1.2 命名约定

好的命名规则能极大地增加可读性和可维护性。同时,对于一个由上百个人共同完成的大项目来说,统一命名约定也是一项必不可少的内容。比较著名的命名规则是 Microsoft 公司的“匈牙利”法,该命名规则的主要思想是“在变量和函数名中加入前缀,以增进人们对程序的理解”。例如,所有的字符变量均以“ch”为前缀,若是指针变量则追加前缀“p”。根据被大多数程序员采纳的共性规则,本节给出程序中的所有标识符(包括变量名、常量名、函数名、类名、结构名、宏定义等)命名时的一般约定。

【规则 1 -1 -33】应用程序的命名形式为“公司名缩写 + 模块名称 + [版本号]”。

【规则 1 -1 -34】子模块的命名规则为每个子模块的名字应该由描述模块功能的 1 ~ 3 个单词组成,每个单词的首字母应大写。在这些单词中可以使用一些较通用的缩写形式。

【规则 1 -1 -35】标识符要采用英文单词或其组合,这样便于记忆和阅读,切忌使用汉语拼音来命名。

说明　标识符应当直观且可以拼读,可望文知义,避免使人产生误解。程序中的英文单词一

般不要太复杂，用词应当准确。

【规则 1-1-36】标识符只能由 26 个英文字母、10 个数字及下画线的一个子集来组成，并严格禁止使用连续的下画线，下画线也不能出现在标识符开始或结尾处（预编译开关除外）。

说明　这样做的目的是为了使程序易读。因为 variable_name 和 variable__name 很难区分，下画线符号“_”若出现在标识符头或结尾，容易与不带下画线的标识符混淆。

【规则 1-1-37】命名规则应尽量与所采用的操作系统或开发工具的风格保持一致。

说明　Windows 应用程序的标识符通常采用“大、小写”混排的方式，如 AddChild，而 UNIX 应用程序的标识符通常采用“小写加下画线”的方式，如 add_child。

【规则 1-1-38】标识符的命名应符合“min-length && max-information”原则。

说明　较短的单词可通过去掉“元音”形成缩写，较长的单词可取单词的前几个字母形成缩写，一些单词有大家公认的缩写，常用单词的缩写必须统一。对于某个系统使用的专用缩写应该在某处做统一说明。

正例　某些单词具有公认的缩写形式，例如，temp 可缩写为 tmp，flag 可缩写为 flg，statistic 可缩写为 stat，increment 可缩写为 inc，message 可缩写为 msg。

常用单词及其缩写形式如表 1.1.1 所示。

表 1.1.1　常用单词缩写

常用词	缩　写	常用词	缩　写
Argument	Arg	Maximum	Max
Buffer	Buf	Message	Msg
Clear	Clr	Minimum	Min
Clock	Clk	Multiplex	Mux
Compare	Cmp	Operating System	OS
Configuration	Cfg	Overflow	Ovf
Context	Ctx	Parameter	Param
Delay	Dly	Pointer	Ptr
Device	Dev	Previous	Prev
Disable	Dis	Priority	Prio
Display	Disp	Read	Rd
Enable	En	Ready	Rdy
Error	Err	Register	Reg
Function	Fnct	Schedule	Sched
Hexadecimal	Hex	Semaphore	Sem
High Priority Task	HPT	Stack	Stk
I/O System	IOS	Synchronize	Sync
Initialize	Init	Timer	Tmr
Mailbox	Mbox	Trigger	Trig
Manager	Mgr	Write	Wr

【规则 1-1-39】程序中不要出现仅靠大、小写字母区分的相似的标识符。

反例

```
int x, X; // 变量 x 与 X 容易混淆
void foo(int x); // 函数 foo 与 FOO 容易混淆
void FOO(float x);
```

【规则 1-1-40】命名中若使用特殊约定或缩写,则应有相应的注释说明。

说明 应该在源文件的开始之处,对文件中使用的缩写或约定,特别是特殊的缩写,进行必要的注释说明。

【规则 1-1-41】自己特有的命名风格,要自始至终保持一致,不可经常变化。

说明 个人的命名风格,在符合所在项目组或产品组的命名规则的前提下,才可使用(即个人命名风格不应与项目组成产品组统一的命名规则矛盾)。

【规则 1-1-42】程序中不要出现标识符完全相同的局部变量和全局变量。

说明 尽管在使用具有完全相同的标识符的局部变量和全局变量时,因其作用域不同不会发生语法错误,但这容易使人误解。

【规则 1-1-43】用正确的反义词组命名具有互斥意义的变量名或相反动作的函数名等。

说明 下面是一些在软件中常用的反义词组。

add/remove	begin/end	create/destroy	insert/delete
first/last	get/release	increment/decrement	put/get
add/delete	lock/unlock	open/close	min/max
old/new	start/stop	next/previous	source/target
show/hide	send/receive	source/destination	cut/paste
up/down			

正例

```
int minValue;
int maxValue;
int SetValue(…);
int GetValue(…);
```

【规则 1-1-44】宏名及常量名都应使用大写字母,用下画线“_”来分割单词。预编译开关的定义应以下画线“_”开始。

正例 如 DISP_BUF_SIZE、MIN_VALUE、MAX_VALUE 等。

【规则 1-1-45】变量名长度应小于 31 个字符,以保持与 ANSI C 标准一致。除局部循环变量名外,不应取单个字符(如 i、j、k 等)作为变量名。

说明　变量,尤其是局部变量,如果用单个字符表示,很容易出错(如 l 误写成 1),在编译时又检查不出来,则有可能增加排错时间。过长的变量名会增加工作量,降低程序的可读性,给修改带来困难,所以应当选择精炼、意义明确的名字,才能简化程序语句,改善对程序功能的理解。

【规则 1-1-46】使用一致的前缀来区分变量的作用域。

说明　变量活动范围前缀规范如下:

g_:全局变量

s_:模块内静态变量

空:局部变量不加范围前缀

【规则 1-1-47】使用一致的小写类型指示符作为前缀来区分变量的类型。

说明　常用的变量类型前缀如表 1.1.2 所示。

表 1.1.2　常用变量类型前缀

前　缀	变量类型	前　缀	变量类型
i	int	b	BOOL
f	float	h	HANDLE
d	double	w	unsigned short 或 WORD
c	char	dw	DWORD 或 unsigned long
uc	unsigned char 或 BYTE	a	数组,array of TYPE
l	long	str	字符串
p	pointer	t	结构类型

以上前缀还可以进一步组合。在进行组合时,数组和指针类型的前缀指示符必须放在变量类型前缀的首位。

【规则 1-1-48】完整的变量名应由“前缀+变量名主体”组成,变量名的主体应当使用名词或者“形容词+名词”的形式,且首字母必须大写。

说明　各种前缀字符可以组合使用,在这种情况下,各前缀顺序为:变量作用域前缀、变量类型前缀。

正例

```
float g_fValue;          //类型为浮点数的全局变量
char  * pcOldChar;       //类型为字符指针的局部变量
```

【规则 1-1-49】函数名由以大写字母开头的单词组合而成，且应当使用动词或者“动词+名词”（动宾词组）的形式。

说明 函数名力求清晰、明了，通过函数名就能够判断函数的主要功能。函数名中不同意义字段之间不要用下画线连接，而应将每个字段的首字母大写以示区分。函数命名采用大、小写字母结合的形式，但专有名词不受限制。

正例

```
DrawBox( ); // 全局函数
box - > Draw( ); // 类的成员函数


void print_record( unsigned int rec_ind );
int input_record( void );
unsigned char get_current_color( void );
```

【规则 1-1-50】除非必要，不要用数字或较奇怪的字符来定义标识符。

反例

```
#define _EXAMPLE_0_TEST_
#define _EXAMPLE_1_TEST_
void set_sls00( BYTE sls );
```

正例

```
#define _EXAMPLE_UNIT_TEST_
#define _EXAMPLE_ASSERT_TEST_
void set_udt_msg_sls( BYTE sls );
```

【规则 1-1-51】在同一软件产品内，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防止编译、链接时产生冲突。

说明 对接口部分的标识符应该有更严格限制，以防止冲突。如可规定接口部分的变量与常量名之前加上其模块名来标识等。

【规则 1-1-52】类名采用大、小写结合的方式，在构成类名的单词之间不应用下画线，类名应以字符“C”开头。类名一般是一个名词或名词短语。

说明 C++ Builder 中的类名首字符为“T”。

正例

```
class CNode; // 类名
class CLeafNode; // 类名
```

【规则 1-1-53】类的数据成员变量名应统一加前缀“m_”（表示成员），这样可以避免数据成员与成员函数的参数同名。

正例

```
void Object::SetValue(int iWidth, int iHeight)
{
    m_iWidth  = iWidth;
    m_iHeight = iHeight;
}
```

【规则 1-1-54】常量名用大写的字母组合而成，并用下画线分割单词。

正例

```
const int MAX = 100;
const int MAX_LENGTH = 100;
```

【规则 1-1-55】结构名、联合名、枚举名由前缀“T_”开头。

【规则 1-1-56】事件名由前缀“EV_”开头。

【规则 1-1-57】尽量避免名字中出现数字编号，如 Value1、Value2 等，除非逻辑上的确需要编号。

说明　这是为了防止程序员偷懒，不肯为命名动脑筋而使用无明确含义的名字（因为用数字编号最省事）。

【规则 1-1-58】标识符前最好不加项目、产品、部门的标识。

说明　这样做的目的是为了代码的可重用性。

【规则 1-1-59】命名时要避免使用国际组织已占用的格式。

说明　已知的被占用的格式如表 1.1.3 所示。

表 1.1.3　已知被占用的格式表

格　式	占用者	格　式	占用者
以双下画线开头	ISO C++、ANSI C	以 LC_开头	ANSI C
包含双下画线	ISO C++	以 SIG[A~Z]开头	ANSI C
以单下画线开头	ISO C++、ANSI C	以 str[a~z]开头	ANSI C
以 E[0~9,A~Z]开头	ANSI C	以 mem[a~z]开头	ANSI C
以 is[a~z]开头	ANSI C	以 wcs[a~z]开头	ANSI C
以 to[a~z]开头	ANSI C	以_t 结尾	POSIX

1.3　注 释 约 定

注释有助于理解代码，有效的注释是指在代码的功能、意图层次上进行注释，提供有用、额外的信息，而不是代码表面意义的简单重复。

【规则 1 - 1 - 60】C 语言的注释符为“/* … */”。C++ 语言中，多行注释采用“/* … */”，单行注释采用“// …”。

说明　注意空格的使用，详见规则 1 - 1 - 26。

【规则 1 - 1 - 61】一般情况下，源程序中有效注释量必须在 20% 以上。

说明　注释的原则是有助于阅读和理解程序，注释不宜太多，也不能太少，注释语言必须准确、易懂、简洁。有效的注释是指对代码的功能、意图等进行注释，提供有用、额外的信息。

【规则 1 - 1 - 62】注释使用中文。

说明　在特殊情况下可以使用英文注释，如工具不支持中文或编制国际化版本程序。

【规则 1 - 1 - 63】文件头部必须进行注释，包括. H 文件、. C 文件、. CPP 文件、. INC 文件、. DEF文件、. CFG 文件(用于编译说明)等。

说明　注释必须列出：版权说明、文件名称、版本号、生成日期、作者、内容摘要、功能、与其他文件的关系、修改日志等，头文件的注释中还应包含对函数功能的简要说明。

正例　下面是文件头部的中文注释：

```
/**********************************************************************
* 版权所有 (C)2004，上海市 XX 股份有限公司。
*
* 文件名称：  // 文件名
* 当前版本：  // 输入当前版本
* 作　　者：  // 输入作者名字及单位
* 完成日期：  // 输入完成日期，例：2004 年 11 月 25 日
* 内容摘要：  //用于简要描述此程序文件完成的主要功能，与其他模块或函数的接
              //口，输出值、取值范围、含义及参数间的控制、顺序、独立或依赖
              //等关系
* 其他说明：  // 其他内容的说明
*
* 主要函数：  //主要函数列表，每条记录应包括函数名及功能简要说明
*     1. ……
* 修改记录 1：// 修改历史记录，包括修改日期、修改者及修改内容
*     修改日期：
```

```
*      版 本 号:
*      修 改 人:
*      修改内容:
* 修改记录2:……
**********************************************************************/
```

下面是文件头部的英文注释:

```
/**********************************************************************
* Copyright (C) 2004,ShangHai XX Corporation.
*
* File Name:          // 文件名(注释对齐)
* Version:            // 输入当前版本
* Author:             // 输入作者名字及单位
* Date:               // 输入完成日期,例:2004-11-25
* Description:        // 简要描述本文件的内容,完成的主要功能
* Others:             // 其他内容的说明
*
* Function List:      //主要函数列表,每条记录应包括函数名及功能简要说明
*      1. …
* History 1:          // 修改历史记录,包括修改日期、修改者及修改内容
*      Date:
*      Version:
*      Author:
*      Modification:
* History 2:…
**********************************************************************/
```

【规则1-1-64】应在函数头部给出关于此函数的注释,包括函数的目的/功能、输入参数、输出参数、返回值、修改信息等。

说明　注释必须列出函数名称、功能描述、输入参数、输出参数、返回值、修改信息等。

正例　下面是函数头部的中文注释:

```
/**********************************************************************
* 函数名称:  // 函数名称
* 功能描述:  // 函数功能、性能等的描述
* 输入参数:  // 输入参数说明,包括每个参数的作用、取值说明及参数间关系
* 输出参数:  // 对输出参数的说明
* 返 回 值:  // 函数返回值的说明
* 其他说明:  // 其他说明
```

```
*  修改日期        版本号      修改人        修改内容
*  ------------------------------------------------
*  2004/11/25      V1.0        XXXX          XXXX
**********************************************************************/
```

下面是函数头部的英文注释：

```
/**********************************************************************
*  Function:        // 函数名称(注释对齐)
*  Description:     // 函数功能、性能等的描述
*  Input:           // 输入参数说明,包括每个参数的作用、取值说明以及参数间关系
*  Output:          // 对输出参数的说明
*  Return:          // 函数返回值的说明
*  Others:          // 其他说明
*  Modify Date      Version      Author       Modification
*  ------------------------------------------------
*  2004/11/25       V1.0         XXXX         XXXX
**********************************************************************/
```

【规则 1-1-65】为了防止头文件被重复引用，应当用 ifndef—define—endif 结构产生预处理块。

· 正例

```
#ifndef GRAPHICS_H // 防止 graphics.h 被重复引用
#define GRAPHICS_H

#include <math.h> // 引用标准库的头文件
…
#include "myheader.h" // 引用非标准库的头文件
…
void Function1(.); // 全局函数声明
…
class Box // 类结构声明
{ …
};
#endif
```

【规则 1-1-66】头文件中只存放“声明”，而不存放“定义”。

说明　在 C++ 语法中，类的成员函数可以在声明的同时被定义，并且自动成为内联函数。这虽然会带来书写上的方便，但却造成了风格不一致，弊大于利。不论函数体有多小，都应将其

成员函数的定义与声明分开。

【规则1-1-67】不提倡使用全局变量,尽量不要在头文件中出现像 extern int value 这样的声明。

【规则1-1-68】在实现文件(即 .CPP 文件)中,全局函数应放在类成员函数的前面。

正例　假设定义文件的名称为 graphics. cpp

```
#include "graphics. h" // 引用头文件
…
// 全局函数的实现体
void Function1(…)
{
    …
}

// 类成员函数的实现体
void Box::Draw(…)
{
    …
}
```

【规则1-1-69】应边写代码边加注释,修改代码的同时修改相应的注释,以保证注释与代码的一致性。不能用陈旧的注释去误导别人,删除无用的注释。

【规则1-1-70】注释是对代码的“提示”,而不是文档。程序中的注释不可喧宾夺主,注释太多了会让人眼花缭乱。注释的花样要少。

【规则1-1-71】如果代码本来就是清楚的,则不必加注释。

反例

```
i++ ; // i 加 1,多余的注释
```

【规则1-1-72】注释的内容要清楚、明了,含义准确,防止注释二义性。

说明　错误的注释不但无益反而有害。

【规则1-1-73】避免在注释语句中使用缩写,尤其是那些不常用的缩写。

说明　在使用缩写时或在使用之前,应对缩写进行必要的说明。

【规则 1 -1 -74】注释应与其描述的代码相近，对代码的注释应放在其上方或右方(对单条语句而言)相邻位置，不可放在语句的下面，如放于上方则需与其上面的代码用空行隔开。

反例 1　如下例子的注释与描述的代码相隔太远。

```
/* 获得子系统索引 */

iSubSysIndex = aData[iIndex].iSysIndex;
```

反例 2　如下例子的注释不应放在所描述的代码下面。

```
iSubSysIndex = aData[iIndex].iSysIndex;
/* 获得子系统索引 */
```

反例 3　如下例子，显得代码与注释过于紧凑。

```
/* 代码段 1 注释 */
// 代码段 1
/* 代码段 2 注释 */
//代码段 2
```

正例　如下书写结构比较清晰

```
/* 获得子系统索引 */
iSubSysIndex = aData[iIndex].iSysIndex;

/* 代码段 1 注释 */
//代码段 1

/* 代码段 2 注释 */
//代码段 2
```

【规则 1 -1 -75】变量注释应放在变量定义后面，并说明变量的用途和取值约定。

正例

```
int status;                    //记录处理状态，0：成功，1：错误
```

【规则 1 -1 -76】包含在{ }中代码块的结束处应加注释，便于阅读。特别是多分支、多重嵌套的条件语句或循环语句。

说明　此时注释可以使用英文，方便查找对应的语句。

正例

```
void main( )
{
    if (…)
```

```
        {
            ...
            while (…)
            {
                ...
            }   /* end of while (…) */      // 指明该条 while 语句结束
        }   /* end of if (…) */             // 指明是哪条语句结束
    } /*   end of void main( ) */            // 指明函数的结束
```

【规则 1-1-77】全局变量要有详细的注释,包括对其功能、取值范围、访问信息及访问时注意事项等的说明。

正例

```
/*
* 变量作用:错误状态码
* 变量范围:0 表示 SUCCESS,1 表示 Table error
* 访问说明:访问的函数以及方法
*/
BYTE g_ucTranErrorCode;
```

【规则 1-1-78】对于所有具有物理含义的变量、常量,如果其命名不是充分自注释的,在声明时都必须加注释,说明其物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。

说明　自注释,即代码本身一读就懂,好像自己在说明自己的逻辑一样。复杂的代码如果实在做不到自注释,应该给出适量的注释。

正例

```
/* active statistic task number */
#define MAX_ACT_TASK_NUMBER 1000

#define MAX_ACT_TASK_NUMBER 1000 /* active statistic task number */
```

【规则 1-1-79】数据结构声明(包括数组、结构、类、枚举等),如果其命名不是充分自注释的,必须加注释。

说明　对数据结构的注释应放在其上方相邻位置,不可放在下面;对结构中的每个域的注释应放在此域的右方。

正例　可按如下形式说明枚举、数组或联合结构。

```
/* sccp interface with sccp user primitive message name */
enum   SCCP_USER_PRIMITIVE
```

```
    {
        N_UNITDATA_IND, /* sccp notify sccp user unit data come */
        N_NOTICE_IND,     /* sccp notify user the No. 7 network can not
                             transmission this message */
        N_UNITDATA_REQ, /* sccp user's unit data transmission request */
    };
```

【规则 1-1-80】注释应与所描述内容进行同样的缩排。

说明 这样可使程序排版整齐,并方便注释的阅读与理解。

反例 如下例子,排版不整齐,阅读不方便。

```
int DoSomething(void)
{
/* 代码段 1 注释 */
        // 代码段 1

/* 代码段 2 注释 */
        // 代码段 2
}
```

正例 如下注释所示,其结构比较清晰。

```
int DoSomething(void)
{
    /* 代码段 1 注释 */
    // 代码段 1

    /* 代码段 2 注释 */
    // 代码段 2
}
```

【规则 1-1-81】对变量的定义和分支语句(条件分支、循环语句等)必须进行注释。

说明 这些语句往往是程序实现某一特殊功能的关键,对于维护人员来说,良好的注释有助于他们更好地理解程序,有时甚至优于看设计文档。

【规则 1-1-82】对于 switch 语句下的 case 语句,如果因为特殊情况需要处理完一个 case 语句后进入另一个 case 语句进行处理,则必须在该 case 语句处理完、进入下一个 case 语句前加上明确的注释。

说明 这样做一方面能防止程序编写者遗漏 break 语句,另一方面也可使维护人员能比较清晰地了解程序流程。

【规则 1-1-83】通过对函数或过程、变量、结构等正确地命名以及合理地组织代码结构，可使代码成为自注释的。

说明　清晰、准确的函数、变量命名，可增加代码的可读性，减少不必要的注释。

【规则 1-1-84】一般而言，对需要删除的代码不建议直接删除，最好用“//”注释起来。

【规则 1-1-85】避免在一行代码或表达式的中间插入注释。

说明　除非必要，不应在代码或表达式中间插入注释，否则容易使代码的可理解性变差。

【规则 1-1-86】应在代码的功能、意图层次上进行注释，提供有用、额外的信息。

说明　注释的目的是解释代码的目的、功能和采用的方法，提供代码以外的信息，可帮助读者理解代码，应避免使用不必要的、重复的注释信息。

反例　如下注释意义不大。

```
/* if receive_flag is TRUE */
if (receive_flag)
```

正例　如下的注释则给出了额外有用的信息。

```
/* if mtp receive a message from links */
if (receive_flag)
```

【规则 1-1-87】使用整行的“*”作为隔离行，可让程序更清晰，可读性更强。

第2章　变量、常量与数据类型

变量、常量和数据类型是编写程序的基础，能否正确地使用它们将直接关系到程序设计的成败。变量包括全局变量、局部变量和静态变量，常量包括数据常量和指针常量，数据类型包括系统的数据类型和自定义数据类型。本章主要说明变量、常量与数据类型使用时必须遵循的规则，关于它们的命名，可参见第1.2节“命名约定”。

2.1　变量与常量

【规则1-2-1】尽量使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串。

说明　C语言用#define来定义常量(称为宏常量)。C++语言除了用#define外，还可以用const来定义常量(称为const常量)。

正例

```
#define      MAX 100          /* C 语言的宏常量 */
const int    MAX = 100;       // C++ 语言的 const 常量
const float  PI = 3.14159;    // C++ 语言的 const 常量
```

【规则1-2-2】在C++程序中只使用const定义常量，而不使用宏常量，即const常量完全取代宏常量。

说明　const不但说明了数据类型，而且声明它是常量，即它的值不能改变。使用const常量而不使用宏常量的优点在于：它创建了一个真正的数据，符号调试程序能够识别const常量而不能识别用#define定义的宏常量。

【规则1-2-3】需要对外公开的常量应放在头文件中，不需要对外公开的常量应放在定义文件的头部。

说明　这样便于管理，可以把不同模块的常量集中存放在一个公共的头文件中。

【规则1-2-4】如果某一常量与其他常量密切相关，应在定义中包含这种关系，而不应给出一些孤立的值。

正例

```
const float RADIUS = 100;
const float DIAMETER = RADIUS * 2;
```

【规则1-2-5】定义全局变量时必须仔细分析，明确其含义、作用、取值范围及与其他全局变量间的关系。

说明　全局变量关系到程序的结构框架，对于全局变量的理解直接影响到能否正确理解整个程序，所以在对全局变量声明的同时，应对其含义、作用及取值范围进行详细地注释和说明，若有必要还应说明与其他变量的关系。

【规则1-2-6】明确全局变量与操作此全局变量的函数或过程的关系。

说明　全局变量与函数的关系包括创建、修改及访问。明确过程操作变量的关系后，将有利于程序的进一步优化、单元测试、系统联调以及代码维护等。可在注释或文档中对这种关系加以说明。

【规则1-2-7】去掉没必要的全局变量，编程时应尽量少用公共变量。

说明　全局变量是增大模块间耦合的原因之一，故应减少没必要的全局变量以降低模块间的耦合度。

【规则1-2-8】当向全局变量传递数据时，要十分小心，防止赋值不合理或越界等现象发生。

说明　对全局变量赋值时，若有必要应进行合法性检查，以提高代码的可靠性、稳定性。

【规则1-2-9】防止局部变量与公共变量同名。

说明　若使用了较好的命名规则，那么此问题可自动消除。

【规则1-2-10】严禁使用未经初始化的变量作为右值。

说明　引用未经初始化的变量可能会产生不可预知的后果，特别是引用未经初始化的指针时经常会导致系统崩溃，需特别注意。声明变量的同时进行初始化，除了能防止引用未经初始化的变量外，还可能生成更高效的机器代码。

【规则1-2-11】一个变量应有且只有一个功能，不能把一个变量用作多种用途。

说明　一个变量只用来表示一个特定功能，不能把一个变量作多种用途，即同一变量取值不同时，其代表的意义也不同。

反例

```
WORD DelRelTimeQue(T_TCB * ptTcb)
{
    WORD wLocate;

    wLocate = 3;
    wLocate = DeleteFromQue(wLocate); // wLocate 具有两种功能
```

```
        return wLocate;
    }
```

正例

```
    WORD DelRelTimeQue( T_TCB * ptTcb )
    {
        WORD wValue;
        WORD wLocate;

        wLocate = 3;
        wValue  = DeleteFromQue( wLocate);

        return wValue;
    }
```

【规则1-2-12】在循环语句与判断语句中，不允许对其他变量进行计算或赋值。

说明　循环语句只完成循环控制功能，判断语句只完成逻辑判断功能，不能完成计算、赋值功能。

反例

```
    do
        {
        //处理语句
    } while ( cInput = GetChar( ) );
```

正例

```
        do
    {
        //处理语句
        cInput = GetChar( );
    } while ( cInput == 0);
```

【规则1-2-13】宏定义中如果包含表达式或变量，表达式和变量必须用圆括号括起来。

说明　在宏定义中，对表达式和变量要使用完备的括号，可以避免可能发生的计算错误。

反例　如下的宏定义表达式都存在一定的隐患：

```
#define HANDLE( A, B) A / B
#define HANDLE( A, B) (A) / (B)
#define HANDLE( A, B) (A / B)
```

对于最后一种情况的宏定义，若 A = 24 + 8，B = 8 * 2；则在实际进行宏替换时，则为HANDLE (24 + 8, 8 * 2)，相当于 (24 + 8 / 8 * 2) = (24 + 1 * 2) = 26 ，与预期的结果不一样。

正例

```
#define HANDLE(A, B) (( A ) / ( B ))
```

若 A =24 +8,B =8 *2;则执行 HANDLE(24 +8, 8 *2)时，相当于 ((24 +8)/(8 *2)) = 2。

【规则 1 -2 -14】使用宏定义多行语句时，必须使用“{ }”把这些语句括起来。

说明　在宏定义中,对多行语句使用花括号,可以避免可能发生的错误。

【规则 1 -2 -15】尽量构造仅有一个模块或函数可以修改、创建的全局变量,而其余有关模块或函数只能访问该全局变量,应防止出现多个模块或函数都可修改、创建同一公共变量的现象。

说明　这样可以减少全局变量操作引起的错误。

正例　在源文件中,可按如下注释形式说明。

```
T _ Student      * g _ ptStudent;
/ * 变量         关系          函数
g _ pStudent     创建          SystemInit( void)
                 修改          无
                 访问          StatScore( const T _ Student  * ptStudent)
                               PrintRec( const T _ Student  * ptStudent)
 * /
```

【规则 1 -2 -16】若变量类型为指针,则在局部分配的空间应在局部释放。

【规则 1 -2 -17】动态全局指针变量空间在程序结束时一定要释放。

【规则 1 -2 -18】函数体内不能分配指针变量空间,也不能将空间指针作为函数参数返回。

【规则 1 -2 -19】所有动态分配的指针变量空间应在对应层次的模块内释放,并且用完马上释放。

【规则 1 -2 -20】不可重复释放相同的指针变量。

【规则 1 -2 -21】对于全局变量应通过统一的函数来访问。

说明　这样可以避免访问全局变量时引起的错误。

正例

```
T _ Student      g _ tStudent;
    T _ Student GetStudentValue( void)
{
T _ Student tStudentValue;
//获取 g _ tStudent 的访问权
```

```
    tStudentValue = g_tStudent;
    //释放 g_tStudent 的访问权
    return tStudentValue;
    }
BYTE SetStudentValue(const T_Student *ptStudentValue)
{
    BYTE ucIfSuccess;
    ucIfSuccess = 0;
    //获取 g_tStudent 的访问权
    g_tStudent = *ptStudentValue;
    //释放 g_tStudent 的访问权
    return ucIfSuccess;
}
```

【规则1-2-22】尽量使用 const 说明常量数据，对于宏定义的常数，必须指出其类型。

反例

```
#define    MAX_COUNT    1000
```

正例

```
const  int  MAX_COUNT = 1000;
#define    MAX_COUNT (int)1000
```

【规则1-2-23】一般不应在语句块内声明局部变量。

【规则1-2-24】使用宏时，不允许参数发生变化。

反例

```
#define SQUARE ((x) * (x))

w = SQUARE( ++value);
```

这个引用将被展开成“w = ((++value) * (++value));”，其中 value 累加了两次，与设计思想不符。

正例

```
#define SQUARE    ((x) * (x))

w = SQUARE(value);
value ++ ;
```

【规则1-2-25】较大的局部变量(2 KB 以上)应声明成静态类型(static)，避免占用太多的堆栈空间。

说明　避免发生堆栈溢出，出现不可预知的软件故障。

【规则 1-2-26】对变量进行赋值时，必须对其值进行合法性检查，防止越界等现象发生。

说明　尤其是对全局变量赋值时，应进行合法性检查，以提高代码的可靠性、稳定性。

【规则 1-2-27】注意变量的有效取值范围，防止表达式出现上溢或下溢。

反例 1

```
unsigned char cIndex = 10;
while( cIndex - - > = 0 )
{
}     //将出现下溢
// 当 cIndex 等于 0 时，再减 1 不会小于 0，而是 0xFF，故程序是一个死循环。
```

反例 2

```
char chr = 127;
chr + = 1;      // 127 为 chr 的边界值，再加 1 将使 chr 上溢到 -128，而不是 128。
```

【规则 1-2-28】防止变量的精度损失。

反例　以下代码将产生精度丢失。

```
#define DELAY _ MILLISECONDS   10000
char time;
time = DELAY _ MILLISECONDS;
WaitTime( time );
```

代码的本意是想产生 10 s 的延时，然而由于 time 为字符型变量，只取 DELAY _ MILLISECONDS 的低字节，高位字节将丢失，结果只产生了 16 ms 的延时。

2.2　类　　型

【规则 1-2-29】结构和联合必须被类型化。

反例

```
struct student
{
    char      acName[ NAME _ SIZE ];
    WORD      wScore;
} * ptStudent;
```

正例

```
typedef struct
{
```

```
    char      acName[NAME_SIZE];
    WORD wScore;
} T_Student;
T_Student * ptStudent;
```

【规则 1-2-30】应使用形式严格定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量。

说明　使用统一的自定义数据类型(如表 1.2.1 所示)，有利于程序的移植。

表 1.2.1　自定义数据类型说明

自定义数据类型	类型说明	类型定义(以 32 位 Windows 系统为例)
VOID	空类型	void
BOOL	逻辑类型 (TRUE 或 FALSE)	unsigned char
BYTE	无符号 8 位整数	unsigned char
CHAR	有符号 8 位整数	signed char
WORD	无符号 16 位整数	unsigned short
SWORD	有符号 16 位整数	signed short
DWORD	无符号 32 位整数	unsigned int
LONG	有符号 32 位整数	signed int
FLOAT	32 位单精度符点数	float
DOUBLE	64 位双精度符点数	double

反例　如下例子(在 DOS 下 BC 3.1 环境中)在移植时可能产生问题。

```
void main()
{
    register int index; // 寄存器变量

    _AX = 0x4000; // _AX 是 BC 3.1 提供的寄存器“伪变量”
    ... // program code
}
```

【规则 1-2-31】结构的功能要单一，是针对一种事务的抽象。

说明　设计结构时应力争使结构作为一种现实事务的抽象，而不是同时代表多种事务。结构中的各元素应代表同一事务的不同侧面，而不应把描述没有关系或关系很弱的不同事务的元素放到同一结构中。

反例　如下结构不太清晰、合理。

```
typedef struct STUDENT_STRU
```

```
{
    unsigned char name[8];          /* student 's name */
    unsigned char age;              /* student 's age */
    unsigned char sex;              /* student 's sex, as follows */
                                    /* 0 - FEMALE; 1 - MALE */
    unsigned char
        teacher_name[8];            /* the student teacher's name */
    unisgned char
        teacher_sex;                /* his teacher sex */
} STUDENT;
```

正例

```
typedef struct TEACHER_STRU
{
    unsigned char name[8];          /* teacher name */
    unisgned char sex;              /* teacher sex, as follows */
                                    /* 0 - FEMALE; 1 - MALE */
} TEACHER;

typedef struct STUDENT_STRU
{
    unsigned char name[8];          /* student 's name */
    unsigned char age;              /* student 's age */
    unsigned char sex;              /* student 's sex, as follows */
                                    /* 0 - FEMALE; 1 - MALE */
    unsigned int teacher_ind;       /* his teacher index */
} STUDENT;
```

【规则 1-2-32】不要设计面面俱到、非常灵活的数据结构。

说明　面面俱到、灵活的数据结构反而容易引起误解和操作困难。

【规则 1-2-33】不同结构间的关系要尽量简单，若两个结构间关系较复杂、密切，那么应合并为一个结构。

说明　两个结构关系复杂时，它们可能反映的是一个事物的不同属性。由于两个结构都是描述同一事物的，那么不如合成一个结构。

反例　如下两个结构的构造不合理。

```
typedef struct PersonOneStruct
{
    BYTE aucName[8];
```

```
    BYTE aucAddr[40];
    BYTE ucSex;
    BYTE ucCity[15];
} T_PersonOne;

typedef struct PersonTwoStruct
{
    BYTE aucName[8];
    BYTE aucAddr[40];
    BYTE ucTel;
} T_PersonTwo;
```

正例

```
typedef struct PersonStruct
{
    BYTE aucName[8];
    BYTE aucAddr[40];
    BYTE ucSex;
    BYTE aucCity[15];
    BYTE ucTel;
} T_Person;
```

【规则1-2-34】结构中元素的个数应适中。若结构中元素个数过多可考虑依据某种原则把元素组成不同的子结构,以减少原结构中元素的个数。

说明 增加结构的可理解性、可操作性和可维护性。

正例 假如认为如上的T_Person结构元素过多,那么可使用如下方式将它进行拆分。

```
typedef struct PersonBaseInfoStruct
{
    BYTE aucName[8];
    BYTE ucSex;
} T_PersonBaseInfo;

typedef struct PersonAddressStruct
{
    BYTE aucAddr[40];
    BYTE aucCity[15];
    BYTE ucTel;
} T_PersonAddress;
```

```
typedef struct PersonStruct
{
    T_PersonBaseInfo tPersonBase;
    T_PersonAddress tPersonAddr;
} T_Person;
```

【规则1-2-35】仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，对于结构中未用的位应明确地给予保留。

说明　合理排列结构中元素顺序，可节省空间并增加可理解性。

反例　如下结构中的位域排列，将占较大空间，可读性也稍差。

```
typedef struct ExampleStruct
{
    BYTE    ucValid: 1;
    T_Person tPerson;
    BYTE    ucSetFlg: 1;
} T_Example;
```

正例　如下形式，不仅可节省字节空间，可读性也变好了。

```
typedef struct ExampleStruct
{
    BYTE ucValid: 1;
    BYTE ucSetFlg: 1;
    BYTE ucOther: 6; // 保留位
    T_Person tPerson;
}T_Example;
```

【规则1-2-36】结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用留有余地(如预留一些空间等)。

说明　软件向前兼容的特性是软件产品是否成功的重要标志之一。如果想使产品具有较好的前向兼容性，那么在产品设计之初就应为以后版本升级留有一定的余地，并且在产品升级时必须考虑前一版本的各种特性。

【规则1-2-37】注意具体语言及编译器处理不同数据类型的原则及有关细节。

说明　如在C语言中，static局部变量将在内存数据区中生成，而非static局部变量将在堆栈中生成。注意这些细节对保证程序的质量非常重要。

【规则1-2-38】合理地设计数据并使用自定义数据类型，尽量减少没有必要的数据类型默认转换与强制转换。

说明　当进行数据类型强制转换时，其数据的意义、转换后的取值等都有可能发生变化，而

这些细节若考虑不周，就很有可能留下隐患。

【规则1-2-39】对自定义数据类型进行恰当的命名，使它成为自注释的，以提高代码可读性。注意其命名方式在同一产品中应统一。

说明　使用自定义类型，可以弥补编程语言提供类型少、信息量不足的缺点，并能使程序清晰、简洁。

正例　下面的声明可使数据类型简洁、明了。

```
typedef unsigned char    BYTE;
typedef unsigned short   WORD;
typedef unsigned int     DWORD;
```

【规则1-2-40】类型声明要避免隐式声明。

说明　尽量用显示声明。

反例

```
main();                          //是 int main( void ) 的隐式表达式
void foo( const iValue); //是 void foo( const int iValue ) 的隐式表达式
```

【规则1-2-41】用 typedef 简化程序中的复杂语法。

说明　当语法非常复杂时（例如函数指针的定义很难读，尤其当该函数比较复杂时），用 typedef 可以大大简化其复杂度，使得阅读、理解都更容易。

正例

```
typedef int ( * CallbackFuncPtr) (int parameter);   //用 typedef 简化函数指针
```

【规则1-2-42】少用 union。

说明　由于 union 成员共用内存空间，所以容易出错，维护困难。而且使用 union 通常意味着要使用非面向对象的方法。

第3章　表达式和基本语句

表达式是语句的一部分，它们是不可分割的。表达式和语句虽然看起来比较简单，但使用时隐患比较多。本章归纳了正确使用表达式和 if、for、while、goto、switch 等基本语句的一些规则与建议。在写表达式和语句的时候要注意运算符的优先级，C/C++ 语言的运算符有数十个，运算符的优先级与结合律如表 1.3.1 所示。

表 1.3.1　运算符的优先级与结合律表

优先级	运算符	结合律
从高到低排列	( ) [ ] -> .	从左至右
	! ~ ++ --（类型）sizeof + - * &	从右至左
	* / %	从左至右
	+ -	从左至右
	<< >>	从左至右
	< <= > >=	从左至右
	== !=	从左至右
	&	从左至右
	∧	从左至右
	|	从左至右
	&&	从左至右
	||	从右至左
	?:	从右至左
	= += -= *= /= %= &= ∧= |= << = >> =	从左至右

【规则 1-3-1】如果代码行中的运算符比较多，应用括号确定表达式的操作顺序，避免使用默认的优先级。

说明　由于要熟记所有运算符的优先级是比较困难的，所以为了防止产生歧义并提高可读性，应当用括号确定表达式的操作顺序。

正例

```
word = (high << 8) | low;

if ((a | b) && (a & c));
```

【规则1-3-2】一条语句只完成一个功能。

说明　复杂的语句阅读起来难于理解，并容易隐含错误。定义变量时，一行只定义一个变量。

反例

```
int iBase, iResult;                    // 一行定义多个变量

iResult = iBase + GetValue(&iBase); // 一条语句实现多个功能，iBase有两种用途
```

正例

```
int iHelp;
int iBase;
int iResult;

iHelp = iBase;
iResult = iHelp + GetValue(&iBase);
```

【规则1-3-3】在表达式中使用括号，可使表达式的运算顺序更清晰。

说明　由于将运算符的优先级与结合律熟记是比较困难的，为了防止产生歧义并提高可读性，即使不加括号时运算顺序不会改变，也应当用括号确定表达式的操作顺序。

反例

```
if (iYear % 4 == 0 && iYear % 100 != 0 || iYear % 400 == 0)
```

正例

```
if (((iYear % 4 == 0) && (iYear % 100 != 0)) || (iYear % 400 == 0))
```

【规则1-3-4】避免表达式中的附加功能，不要编写太复杂的复合表达式。

说明　带附加功能的表达式难于阅读和维护，它们常常导致错误。

反例

```
iResult = (aiVar[1] = aiVar[2] + aiVar[3]++) + ++aiVar[4];
```

正例

```
aiVar[1] = aiVar[2] + aiVar[3];
aiVar[4]++;
iResult = aiVar[1] + aiVar[4];
```

```
aiVar[3] ++ ;
```

【规则 1 -3 -5】不要有多用途的复合表达式。

反例

```
d = (a = b + c) + r ;
```

正例　该表达式既求 a 值又求 d 值,应该拆分为两个独立的语句。

```
a = b + c;
d = a + r;
```

【规则 1 -3 -6】不要把程序中的复合表达式与“真正的数学表达式”相混淆。

反例

```
if (a < b < c);      // a < b < c 是数学表达式而不是程序表达式
```

正例

```
if ((a<b) && (b<c));
```

【规则 1 -3 -7】不可将布尔变量和逻辑表达式直接与 TRUE、FALSE 或者 1、0 进行比较。

说明　TURE 和 FALSE 的定义值是和语言环境相关的,且可能会被重定义。

反例　设 bFlag 是布尔类型的变量

```
if (bFlag == TRUE)
if (bFlag == 1)
if (bFlag == FALSE)
if (bFlag == 0)
```

正例　设 bFlag 是布尔类型的变量

```
if (bFlag)        // 表示 flag 为真
if (! bFlag)      // 表示 flag 为假
```

【规则 1 -3 -8】在条件判断语句中,当整型变量与 0 比较时,不可模仿布尔变量的风格,应当将整型变量用“ ==”或“ ! =”直接与 0 比较。

反例

```
if (iValue) // 会让人误解 iValue 是布尔变量


if (!iValue)
```

正例

```
if (iValue ==0)
```

```
if (iValue ! = 0)
```

【规则 1 -3 -9】不可将浮点变量用“ == ”或“ ! = ”与任何数字比较。

说明　无论是 float 还是 double 类型的变量,都有精度限制。所以一定要避免将浮点变量用“ == ”或“ ! = ”与数字比较,应该转化成“ >= ”或“ <= ”形式。

反例

```
if (fResult == 0.0)        // 隐含错误的比较
```

正例

```
if ((fResult >= - EPSINON) && (fResult <= EPSINON))
//EPSINON 是允许的误差,即精度
```

【规则 1 -3 -10】应当将指针变量用“ == ”或“ ! = ”与 NULL 比较。

说明　指针变量的零值是“空”(记为 NULL),即使 NULL 的值与 0 相同,但是两者意义不同。

反例

```
if (pHead ==0)          // 容易让人误解 pHead 是整型变量
if (pHead ! = 0)
或者
if (pHead)              // 容易让人误解 pHead 是布尔变量
if (! pHead)
```

正例

```
if (pHead == NULL)  // pHead 与 NULL 显式比较,强调 pHead 是指针变量
if (pHead ! = NULL)
```

【规则 1 -3 -11】在 switch 语句中,每一个 case 分支必须使用 break 结尾,最后一个分支必须是 default 分支。

说明　避免漏掉 break 语句造成程序错误。同时保持程序简洁。对于多个分支处理情况相同时可以共用一个 break,但是要用注释加以说明。

正例

```
switch (iMessage)
{
    case SPAN_ON:
    {
        //处理语句
        break;
```

```
        }
        case SPAN_OFF:
        {
            //处理语句
            break;
        }
        default:
        {
            //处理语句
        }
    }
```

【规则 1 -3 -12】不要忘记最后的 default 分支。

说明　即使程序真的不需要 default 处理,也应该保留语句“default : break;”,这样做并非多此一举,而是为了养成一种良好的编程习惯。

【规则 1 -3 -13】不可在 for 循环体内修改循环变量,防止 for 循环失去控制。

【规则 1 -3 -14】循环嵌套次数不应大于 3。

【规则 1 -3 -15】do while 语句和 while 语句仅使用一个条件。

说明　这样可保持程序简洁。如果需要判断的条件较多,建议用临时布尔变量先计算出其值。

正例

```
BOOLEAN bCondition;

do
{
    ...
    bCondition = ((tAp[iPortNo].bStateAcpActivity != PASSIVE)
                || (tAp[iPortNo].bStateLacpActivity != PASSIVE))
                && (abLacpEnabled[iPortNo])
                && (abPortEenabled[iPortNo])

} while (bCondition);
```

【规则 1-3-16】当 switch 语句的分支比较多时,采用数据驱动方式。

说明 当 switch 语句中 case 语句比较多时,会降低程序的效率。

正例

```
extern void TurnState(void);
extern void SendMessage (void);
...
void ( * StateChange[20])() = {TurnState, SendMessage, NULL, TurnState··· };
...
if (StateChange[iState])
{
    ( * StateChange[iState])();
}
```

【规则 1-3-17】如果循环体内存在逻辑判断,并且循环次数很多,宜将逻辑判断移到循环体的外面。

说明 下面两个示例中,反例比正例多执行了 NUM -1 次逻辑判断。并且由于前者总要进行逻辑判断,使得编译器不能对循环进行优化处理,降低了效率。如果 NUM 非常大,最好采用正例的写法,可以提高效率。

反例

```
const int NUM = 100000;
for (i = 0; i < NUM; i ++)
{
    if (bCondition)
    {
        DoSomething();
    }
    else
    {
        DoOtherthing();
    }
}
```

正例

```
const int NUM = 100000;
if (bCondition)
{
    for (i = 0; i < NUM; i ++)
```

```
        {
            DoSomething( );
        }
    }
    else
    {
        for (i = 0; i < NUM; i ++ )
        {
            DoOtherthing( );
        }
    }
```

【规则 1 -3 -18】for 语句的循环控制变量的取值应采用“半开半闭区间”写法。

说明　这样做更能适应 C 语言数组的特点,C 语言的下标属于一个“半开半闭区间”。

反例

```
int aiScore[ NUM ];
...
for (i = 0; i <= NUM -1; i ++ )
{
    printf( "% d\n" ,aiScore[ i ]);
}
```

正例

```
int aiScore[ NUM ];
...
for (i = 0; i < NUM; i ++ )
{
    printf( "% d\n" ,aiScore[ i ])
}
```

相比之下,正例的写法更加直观,尽管两者的功能是相同的。

【规则 1 -3 -19】在进行“ == ”比较时,将常量或常数放在“ == ”号的左边,以便在将“ = = ”误写成“ = ”时,编译器可以发现错误。

说明　可以采用这种方式,让编译器去发现错误。

正例

```
if (NULL == pTail)
if (0 == iSum)
```

示例中有意把 pTail 和 NULL 颠倒,以避免将“==”误写成“=”。编译器认为 if (pTail = NULL) 是合法的(但实际上达不到预期的目的,该句含义是将 NULL 赋值给 pTail,然后再判别 pTail 是否为真),但是会指出 if (NULL = pTail)是错误的,因为 NULL 不能被赋值。

【规则 1-3-20】在多重循环中,如果有可能,应当将最长的循环放在最内层,最短的循环放在最外层。

说明 这样可以减少 CPU 跨越循环层的次数。也就是说,在多重循环中,应将最忙的循环放在最内层。

反例 长循环在最外层,故效率低。

```
for (row = 0; row < 100; row ++)
{
    for ( col = 0; col < 5; col ++ )
    {
        sum = sum + a[row][col];
    }
}
```

正例 效率高,因为长循环在最内层。

```
for (col = 0; col < 5; col ++ )
{
    for (row = 0; row < 100; row ++ )
    {
        sum = sum + a[row][col];
    }
}
```

【规则 1-3-21】建议少用、慎用 goto 语句。

说明 自从提倡结构化设计以来,goto 就成了有争议的语句。首先,由于 goto 语句可以灵活跳转,如果不加限制,它就会破坏结构化设计风格。其次,goto 语句经常带来错误或隐患。它可能跳过了某些对象的构造、变量的初始化、重要的计算等语句。

反例

```
goto state;
String s1, s2; // 被 goto 跳过
int sum = 0; // 被 goto 跳过
   ...
state:
   ...
```

如果编译器不能发觉此类错误，每执行一次 goto 语句都可能留下隐患。但实事求是地说，错误是程序员自己造成的，不是 goto 语句的过错。

【规则 1-3-22】建议循环、判断语句的程序块部分用花括号括起来，即使只有一条语句。

反例

```
if( bCondition == TRUE )
    bFlag = YES;
```

正例　建议按以下方式书写，便于代码的修改、增删。

```
if( bCondition == TRUE )
{
    bFlag = YES;
}
```

【规则 1-3-23】在 switch 语句中将经常性的处理放在前面。

第4章　函数和过程

函数是C /C ++ 程序的基本功能单元,其重要性不言而喻。如何编写出正确、高效、易维护的函数是软件编码质量控制的关键。一个函数包括函数头、函数名、函数体、参数和返回值。其中函数头的编写参见第1.3节“注释约定”,函数名参见第1.2节“命名约定”,本章着重描述作为接口要素的参数和返回值、函数体的实现以及函数相互之间的调用关系。

函数接口的两个要素是参数和返回值。C语言中,函数的参数和返回值的传递方式有两种:值传递(pass by value)和指针传递(pass by pointer)。C ++ 语言中多了引用传递(pass by reference)方式。

4.1　参数规则

【规则1-4-1】参数的书写要完整,不要贪图省事只写参数的类型而省略参数名字。如果函数没有参数,则用void填充。

说明　函数在说明的时候,可以省略参数名。但是为了提高代码的可读性,要求不能省略。

反例

```
void SetValue(int, int);
float GetValue();
```

正例

```
void SetValue(int iWidth, int iHeight);
float GetValue(void);
```

【规则1-4-2】参数命名要恰当,顺序要合理。

反例　函数声明为:

```
void StringCopy(char * str1, char * str2);
```

我们很难搞清楚究竟是把str1拷贝到str2中,还是刚好倒过来。

正例　函数声明为:

```
void StringCopy(char * strSource, char * strDestination);
```

参数的顺序要遵循程序员的习惯。一般地,应将目的参数放在前面,源参数放在后面。

【规则1-4-3】如果参数是指针,且仅作输入用,则应在类型前加“const”。

说明　防止该指针在函数体内被意外修改。

正例

```
int GetStrLen(const char * pcString);
```

【规则 1 -4 -4】当结构变量作为参数时,应传送结构的指针而不传送整个结构体,并且不得修改结构中的元素,但用作输出时除外。

说明　一个函数被调用的时候,形参会被一个个压入被调函数的堆栈中,在函数调用结束以后再弹出。一个结构所包含的变量往往比较多,直接以一个结构为参数,压栈、出栈的内容就会太多,不但占用堆栈空间,而且影响代码执行效率,如果使用不当还可能导致堆栈的溢出。如果使用结构的指针作为参数,因为指针的长度是固定不变的,结构的大小就不会影响代码执行的效率,也不会过多地占用堆栈空间。

【规则 1 -4 -5】避免函数有太多的参数,参数个数应尽量控制在 5 个以内。

说明　如果参数太多,在使用时容易将参数类型或顺序搞错,而且调用的时候也不方便。如果参数的确比较多,而且输入的参数相互之间的关系比较紧密,不妨把这些参数定义成一个结构,然后把结构的指针当成参数输入。

【规则 1 -4 -6】参数的顺序要合理。

说明　参数的顺序要遵循程序员的习惯。如输入参数放在前面,输出参数放在后面等。

正例

```
int RelRadioChan(const T_RelRadioChanReq * ptReq, T_RelRadioChanAck * ptAck);
```

【规则 1 -4 -7】尽量不要使用类型和数目不确定的参数。

说明　对于参数个数可变的函数调用,编译器不作类型检查和参数检查。

【规则 1 -4 -8】避免使用布尔类型的参数。

说明　一方面因为布尔类型的参数值无意义,TRUE/FALSE 的含义是非常模糊的,在调用时很难知道该参数到底传达的是什么意思;其次布尔类型的参数值不利于扩充。

【规则 1 -4 -9】防止将函数的参数作为工作变量。

说明　将函数的参数作为工作变量,有可能错误地改变参数内容,所以很危险。对必须改变的参数,最好先用局部变量参与函数体的运算,最后再将该局部变量的内容赋给该参数。

反例

```
void sum_data( unsigned int num, int *data, int *sum )
{
    unsigned int count;

    *sum = 0;
    for (count = 0; count < num; count++)
    {
        *sum += data[count]; // sum 成了工作变量,不太好
    }
}
```

正例

```
void sum_data( unsigned int num, int *data, int *sum )
{
    unsigned int count ;
    int sum_temp;

    sum_temp = 0;
    for (count = 0; count < num; count++)
    {
        sum_temp += data[count];
    }

    *sum = sum_temp;
}
```

【规则1-4-10】非调度函数应减少或防止控制参数,尽量只使用数据参数。

说明 本规则是防止函数间的控制耦合。调度函数是指根据输入的消息类型或控制命令,来启动相应的功能实体(即函数或过程),而本身并不完成具体功能。控制参数是指能改变函数功能或行为的参数,即函数要根据此参数来决定具体怎样工作。非调度函数的控制参数增加了函数间的控制耦合,很可能使函数间的耦合度增大,并使函数的功能不惟一。

反例 如下函数构造不太合理。

```
int add_sub( int a, int b, unsigned char add_sub_flg )
{
    if (add_sub_flg == INTEGER_ADD)
    {
        return (a + b);
    }
```

```
        else
        {
            return (a - b);
        }
    }
```

正例　不如分为如下两个函数,这样使每个函数的功能都很清晰。

```
int add( int a, int b )
{
    return (a + b);
}

int sub( int a, int b )
{
    return (a - b);
}
```

【规则 1 -4 -11】检查函数所有参数输入的有效性。

说明　可直接检查或使用断言进行检查,尤其是指针参数。只在本模块内使用的函数可不检查。

【规则 1 -4 -12】检查函数所有非参数输入的有效性,如数据文件、公共变量等。

说明　函数的输入主要有两种:一种是参数输入;另一种是全局变量、数据文件的输入,即非参数输入。函数在使用输入之前,应进行必要的检查。

【规则 1 -4 -13】声明函数原型时,对于数组型参数,不要声明为指针,以维护函数接口的清晰性。

假设函数 SortInt( )完成的功能是对一组整数排序,接收的参数是一整数数组及数组中的元素个数,则有以下两例。

反例

```
void SortInt(int num, int *data);
```

正例

```
void SortInt(int num, int data[ ]);
```

4.2　返回值规则

【规则 1 -4 -14】不要省略返回值的类型，如果函数没有返回值，那么应声明为 void 类型。

说明　C 语言中，凡不加类型说明的函数，一律自动按整型处理。如果不注明类型，容易被误解为 void 类型，产生不必要的麻烦。

C ++ 语言有很严格的类型安全检查，不允许上述情况发生。由于 C ++ 程序可以调用 C 函数，为了避免混乱，规定任何 C/ C ++ 函数都必须有类型。

【规则 1 -4 -15】对于有返回值的函数，每一个分支都必须有返回值。

说明　为了保证对被调用函数返回值的判断，有返回值的函数中的每一个退出点都需要有返回值。

【规则 1 -4 -16】如果返回值表示函数运行是否正常，规定 0 为正常退出，不同的非 0 值标识不同的异常退出情况。避免使用 TRUE 或 FALSE 作为返回值。

反例

```
BOOL SubFunction(void);
```

正例

```
int SubFunction(void);
```

【规则 1 -4 -17】函数名字与返回值类型在语义上不可冲突。

反例　违反这条规则的典型代表是 C 标准库函数 getchar。

```
char c;
c = getchar();
if (c == EOF)
...
```

按照 getchar 名字的意思，将变量 c 声明为 char 类型是很自然的事情。但不幸的是 getchar 的确不是 char 类型，而是 int 类型，其原型是："int getchar(void);"。

【规则 1 -4 -18】有时候函数原本不需要返回值，但为了增加灵活性，如支持链式表达，可以附加返回值。

正例　字符串拷贝函数 strcpy 的原型：

```
char * strcpy(char * strDest,const char * strSrc);
```

strcpy 函数将 strSrc 拷贝至输出参数 strDest 中，同时函数的返回值也是 strDest。这样做并非多此一举，可以获得如下灵活性：

```
char str[20];
int length = strlen( strcpy(str, "Hello World") );
```

【规则 1 -4 -19】函数的返回值应清楚、明了，避免遗漏错误情况的处理。

说明　函数的每种出错返回值的意义都应清晰、明了、准确，防止出现误用返回码或忽视错误等情况。

【规则 1 -4 -20】除非必要，最好不要把与函数返回值类型不同的变量以编译系统默认的转换方式或强制的转换方式作为返回值返回。

4.3　内部实现规则

函数体的实现并不是随心所欲的，而是有一定的规矩可循。不但要仔细检查入口参数的有效性和精心设计返回值，还要保证函数的功能单一，具有很高的功能内聚性，尽量减少函数之间的耦合，方便调试和维护。

【规则 1 -4 -21】在函数体的“出口处”，应对 return 语句的正确性和效率进行检查。

说明　如果函数有返回值，那么函数的“出口处”是 return 语句。如果 return 语句写得不好，函数要么出错，要么效率低下。

① return 语句不可返回指向“栈内存”的指针或者引用，因为该内存在函数体结束时会被自动销毁。

反例

```
char * Func(void)
{
    char str[ ] = "hello world"; // str 的内存位于栈上
    ...
    return str; // 将导致错误
}
```

② 应明确返回的究竟是值、指针，还是引用。

③ 如果函数返回值是一个对象，要考虑 return 语句的效率。

【规则 1 -4 -22】明确函数功能，精确（而不是近似）地实现函数设计。

【规则 1 -4 -23】编写可重入函数时，应注意局部变量的使用（如编写 C/C ++ 语言的可重入函数时，应使用 auto 即缺省态局部变量或寄存器变量）。

说明　可重入性是指函数可以被多个任务进程调用。在多任务操作系统中，函数是否具有可重入性是非常重要的，因为这是多个进程可以共用此函数的必要条件。另外，编译器是否提供可重入函数库与它所服务的操作系统有关，只有操作系统是多任务时，编译器才有可能提供可重入函数库。如 DOS 下 BC 和 MSC 等就不具备可重入函数库，因为 DOS 是单用户、单任务操作系统。

编写 C/C ++ 语言的可重入函数时，不应使用 static 局部变量，否则必须经过特殊处理，才能使函数具有可重入性。

一个可重入的函数简单来说，就是可以被中断的函数，它主要在多任务环境中使用。也就是说，可以在这个函数执行的任何时候中断它的运行，在操作系统的调度下去执行另外一段代码而不会出现错误。

而不可重入的函数由于使用了一些系统资源，比如全局变量区、中断向量表等，所以它如果被中断，可能出现问题，所以这类函数是不能运行在多任务环境下的。

【规则 1 -4 -24】编写可重入函数时，若使用全局变量，则应通过关中断、信号量（即 P、V 操作）等手段对其加以保护。

说明　若对所使用的全局变量不加以保护，则此函数就不具有可重入性，即当多个进程调用此函数时，很有可能使有关全局变量变为不可知状态。

反例　假设 Exam 是 int 型全局变量，函数 Squre_Exam 返回 Exam 平方值。那么如下函数不具有可重入性。

```
unsigned int example( int para )
{
    unsigned int temp;

    Exam = para; // ①
    temp = Square_Exam( );//②

    return temp;
}
```

此函数若被多个进程调用，则其结果可能是未知的。假设标记为①的语句刚执行完，另外一个使用本函数的进程可能正好被激活，那么当新激活的进程执行到此函数时，将使 Exam 被赋以另一个不同的 para 值，所以当控制重新回到标记为②的语句后，计算出的 temp 很可能不是预想

中的结果。

正例　对以上函数应作如下改进：

```
unsigned int example( int para )
{
    unsigned int temp;

    //申请信号量操作
    /* 若申请不到信号量,说明另外的进程正处于给 Exam 赋值并计算其平方过程中(即正在使用此信号量),本进程必须等待其释放信号量后,才可继续执行。若申请到信号量,则可继续执行,但其他进程必须等待本进程释放信号量后,才能再使用本信号量。*/
    Exam = para;
    temp = Square_Exam( );
    //释放信号量操作
    return temp;
}
```

【规则 1 -4 -25】函数的规模应尽量限制在 200 行以内。

说明　注意,计算时不包括注释和空格行。冗长的函数不利于调试,可读性差。

【规则 1 -4 -26】函数的功能要单一,不要设计多用途的函数。

说明　多用途的函数往往通过在输入参数中设置有一个控制参数,来实现根据不同的控制参数产生不同的功能。这种方式增加了函数之间的控制耦合性,而且在函数调用的时候,调用相同的一个函数却产生不同的效果,降低了代码的可读性,也不利于代码调试和维护。

反例　如果把这两个函数合并在一个函数中,通过控制参数 ucAddOrSubFlg 决定结果,则不可取。

```
int AddOrSub(int iParaOne, int iParaTwo, unsigned char ucAddOrSubFlg)
{
if (INTEGER_ADD == ucAddOrSubFlg) // 参数标记为“求和”
{
    return (iParaOne + iParaTwo);
}
else
{
    return (iParaOne - iParaTwo);
}
```

```
}
```

正例　以下两个函数功能清晰：

```
int Add(int iParaOne, int iParaTwo)
{
    return (iParaOne + iParaTwo);
}

int Sub(int iParaOne, int iParaTwo)
{
    return (iParaOne - iParaTwo);
}
```

【规则 1-4-27】尽量避免函数带有“记忆”功能。函数的输出应该具有可预测性，即相同的输入应当产生相同的输出。

说明　带有“记忆”功能的函数，其行为可能是不可预测的，因为它的行为可能取决于某种“记忆状态”。这样的函数既不易理解，又不利于测试和维护。在 C/C ++ 语言中，函数的 static 局部变量是函数的“记忆”存储器。建议尽量少用 static 局部变量，除非确实需要。

【规则 1-4-28】为简单功能编写函数。

说明　虽然为仅用一两行就可完成的功能去编函数好像没有必要，但使用函数可增加程序可读性，亦可方便维护、测试。

反例　如下语句的功能不很明显。

```
iMaxValue = (iParaOne > iParaTwo) ? iParaOne: iParaTwo;
```

正例　如下程序段显得很清晰。

```
int Max(int iParaOne, int iParaTwo)
{
    int iMaxValue;
    iMaxValue = (iParaOne > iParaTwo) ? iParaOne: iParaTwo;
    return iMaxValue;
}
```

【规则 1-4-29】函数功能应明确，防止把没有关联的语句放到一个函数中。

说明　防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便，同时也使函数或过程的功能不明确。使用随机内聚函数，常常容易出现在一种应用场合需要改进此函

数,而另一种应用场合又不允许这种改进的情况,从而陷入困境。

反例　矩形的长、宽与点的坐标之间没有直接的关系,故以下函数是随机内聚。

```
void InitVar(void)
{
    // 初始化矩形的长与宽
    tRect.wLength = 0;
    tRect.wWidth = 0;

    // 初始化点的坐标
    tPoint.wX = 10;
    tPoint.wY = 10;
}
```

正例　矩形的长、宽与点的坐标之间没有直接的关系,应该在不同的函数中实现。

```
void InitRect(void)
{
    // 初始化矩形的长与宽
    tRect.wLength = 0;
    tRect.wWidth = 0;
}

void InitPoint(void)
{
    // 初始化点的坐标
    tPoint.wX = 10;
    tPoint.wY = 10;
}
```

【规则1-4-30】尽量不要编写依赖于其他函数内部实现的函数。

说明　此规则是函数独立性的基本要求。由于目前大部分高级语言都是结构化的,所以通过具体语言的语法要求与编译器功能,基本就可以防止这种情况发生。但在汇编语言中,由于其灵活性,很可能使函数出现这种情况。

【规则1-4-31】在调用函数填写参数时,应尽量减少没有必要的默认数据类型转换或强制数据类型转换。

说明　因为数据类型转换或多或少存在危险。

【规则 1 -4 -32】避免函数中不必要的语句，防止程序中出现垃圾代码。

说明　程序中的垃圾代码不仅占用额外的空间，而且还常常影响程序的功能与性能，很可能给程序的测试、维护等造成不必要的麻烦。

【规则 1 -4 -33】功能不明确且较小的函数，特别是仅有一个上级函数调用它时，应考虑把它合并到上级函数中，而不必独立存在。

说明　模块中函数划分得过多，一般会使函数间的接口变得复杂。所以过小的函数，特别是扇入很低的或功能不明确的函数，不应独立存在。

【规则 1 -4 -34】改进模块中函数的结构，降低函数间的耦合度，并提高函数的独立性以及代码可读性、效率和可维护性。

说明　优化函数结构时，要遵守以下原则：

① 不能影响模块功能的实现。

② 仔细考查模块或函数出错处理及模块的性能要求并进行完善。

③ 通过分解或合并函数来改进软件结构。

④ 考查函数的规模，其中规模过大的函数要进行分解。

⑤ 降低函数间接口的复杂度。

⑥ 不同层次的函数调用要有较合理的扇入、扇出。

⑦ 函数结果应可预测。

⑧ 提高函数内聚性（单一功能的函数内聚最高）。

对初步划分后的函数结构应进行改进、优化，使之更为合理。

【规则 1 -4 -35】在多任务操作系统的环境下编程，要注意函数可重入性的构造。

【规则 1 -4 -36】对于提供了返回值的函数，在引用时最好使用其返回值。

【规则 1 -4 -37】在一个过程（函数）中，若多次引用名称较长的变量（一般是结构的成员），则应用一个宏来代替。

说明　这样可以增加编程效率和程序的可读性。

正例　若在某过程中较多引用 TheReceiveBuffer[FirstSocket]. byDataPtr，则可以通过以下宏定义来代替：

```
# define pSOCKDATA TheReceiveBuffer[ FirstScoket ]. byDataPtr
```

【规则 1 -4 -38】避免相同的代码段在多个地方出现。

说明　当某段代码需在不同的地方重复使用时，应根据代码段的规模大小使用函数调用或宏调用的方式来代替。这样对该代码段的修改就可在一处完成，从而增强了代码的可维护性。

【规则 1 -4 -39】使用专门的初始化函数对所有的公共变量进行初始化。

4.4　函数调用规则

【规则 1 -4 -40】必须对所调用函数的错误返回值进行处理。

说明　函数返回错误，往往是因为输入的参数不合法，或者此时系统已经出现了异常。如果不对错误返回值进行必要的处理，会导致错误的扩大，甚至导致系统的崩溃。

反例　假设在程序中定义了一个函数：

```
int DbAccess( WORD wEventNo, T_InPara * ptInParam, T_OutPara * ptOutParam);
```

对上面定义的函数进行如下的处理就不合适。

```
DbAccess( EV_GETRADIOCHANNEL, ptReq, ptAck);
//正常处理
```

正例　在引用该函数的时候应该如下处理：

```
int iResult;
iResult = DbAccess( EV_GETRADIOCHANNEL, ptReq, ptAck);
switch (iResult)
{
    case NO_CHANNEL:      // 无可用无线资源
    {
        //异常处理
        break;
    }
case CELL_NOTFOUND: // 小区未找到
{
    //异常处理
    break;
}
default:
{
    //其他处理
```

```
        }
    }
    //正常处理
```

【规则 1-4-41】减少函数本身或函数间的递归调用。

说明　递归调用特别是函数间的递归调用(如 A→B→C→A),将影响程序的可读性;递归调用一般都占用较多的系统资源(如栈空间);且对程序的测试有一定影响。故除非为某些算法或功能的实现方便,应减少不必要的递归调用。

对于前台软件,为了系统的稳定性和可靠性,往往规定了进程的堆栈大小。如果采用了递归算法,收敛的条件又往往难以确定,很容易使进程的堆栈溢出,破坏系统的正常运行;另外,由于无法确定递归的次数,也就降低了系统的稳定性和可靠性。

【规则 1-4-42】设计高扇入、合理扇出的函数。

说明　扇出是指一个函数直接调用(控制)其他函数的数目,而扇入是指有多少上级函数调用它。

扇出过大,表明函数过分复杂,需要控制和协调过多的下级函数;而扇出过小,如总是 1,则表明函数的调用层次可能过多,这样不利于程序阅读和分析函数结构,并且程序运行时会对系统资源如堆栈空间等造成压力。函数较合理的扇出(调度函数除外)通常是 3~5。扇出太大,一般是由于缺乏中间层次,可适当增加中间层次的函数。扇出太小,可把下级函数进一步分解成多个函数,或合并到上级函数中。当然分解或合并函数时,不能改变要实现的功能,也不能违背函数间的独立性。

扇入越大,表明使用此函数的上级函数越多,这样的函数使用效率高,但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。

较好的软件结构通常是顶层函数的扇出较高,中层函数的扇出较少,而底层函数则扇入到公共模块中。

第5章　内存和指针

5.1　内存使用规则

【规则1-5-1】在程序编制之前，必须了解编译系统的内存分配方式，特别是编译系统对不同类型的变量的内存分配规则，如局部变量在何处分配、静态变量在何处分配等。

【规则1-5-2】防止内存操作越界。

说明　内存操作主要是指对数组、指针、内存地址等的操作，内存操作越界是软件系统主要错误之一，后果往往非常严重，所以在进行这些操作时一定要仔细。

反例

```
const int MAX_USE_NUM = 10                                    // 用户号为1~10
unsigned char aucLoginFlg[MAX_USR_NUM] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
void ArrayFunction(void)
{
    unsigned char ucUserNo;
    for (ucUserNo = 1; ucUserNo < 11; ucUserNo ++ )           // 11已经越界了
    {
        aucLoginFlg[User_No] = ucUserNo;
        ...
    }
}
```

正例

```
const int MAX_USE_NUM = 10                                    // 用户号为1~10
unsigned char aucLoginFlg[MAX_USR_NUM] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

void ArrayFunction(void)
{
    unsigned char ucUserNo;
    for (ucUserNo = 0; ucUserNo < MAX_USE_NUM; ucUserNo ++ )
    {
        aucLoginFlg[ucUser_No] = ucUserNo;
        ...
```

```
}
}
```

【规则 1-5-3】必须对动态申请的内存做有效性检查,并进行初始化;动态内存的释放必须和分配成对以防止内存泄漏,释放后应将内存指针置为 NULL。

说明 对嵌入式系统,通常内存是有限的,申请内存的操作可能会失败,如果不检查就对该指针进行操作,可能出现异常,而且这种异常不是每次都出现,比较难定位。

指针释放后,该指针可能还是指向原有的内存块,也有可能改变指向,变成一个“野指针”,一般用户不会对它再进行操作,但用户失误情况下对它的操作可能导致程序崩溃。

正例

```
MemmoryFunction(void)
{
    unsigned char *pucBuffer = NULL;
    pucBuffer = GetBuffer(sizeof(DWORD));
    if (NULL != pucBuffer)                    // 申请的内存指针必须进行有效性验证
    {
        // 申请的内存使用前必须进行初始化
        memset(pucBuffer, 0xFF, sizeof(DWORD));
    }
    ...
    FreeBuffer(pucBuffer);                    // 申请的内存使用完毕必须释放
    pucBuffer = NULL;                         // 申请的内存释放后指针置为空
    ...
}
```

【规则 1-5-4】不使用 realloc( )。

说明 调用 realloc 可对一个内存块进行扩展,但会改变内存块中原有内容的存储位置。realloc 函数既要调用 free,又要调用 malloc。执行时究竟调用哪个函数,取决于是要缩小还是扩大相应内存块的大小。

【规则 1-5-5】变量在使用前应初始化,防止未经初始化的变量被引用。

说明 变量在定义之后、初始化之前,其值会因编译系统的不同而不确定。有些系统会初始化为 0,而有些不是。

【规则 1 -5 -6】由于内存总量是有限的,软件系统各模块应约束自己的代码,尽量少占用系统内存。

【规则 1 -5 -7】在大程序中,为了保证高可靠性,一般不使用动态分配内存的方法。

【规则 1 -5 -8】在往一个内存区连续赋值之前(memset,memcpy,…),应确保内存区的大小能够容纳所赋的数据量。

【规则 1 -5 -9】尽量使用 memmove(　)方法代替 memcpy(　)方法。

说明　在源、目的内存区域发生重叠的情况下,如果使用 memcpy(　)方法可能导致重叠区的数据被覆盖。

【规则 1 -5 -10】使用正确的内存分配方式。

说明　内存分配方式有 3 种:

① 从静态存储区域分配:内存在程序编译的时候就已经分配好,这块内存在程序的整个运行期间都存在。例如为全局变量、static 变量分配的内存。

② 在栈上创建:在函数执行时,函数内局部变量的存储单元都可以在栈上创建,函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中,效率很高,但是分配的内存容量有限。

③ 从堆上分配:亦称动态内存分配。程序在运行的时候用 malloc 或 new 申请任意的内存,程序员自己负责在何时用 free 或 delete 释放内存。动态内存的生存期由程序员决定,使用非常灵活,但使用时需注意的问题也最多。

【规则 1 -5 -11】内存分配后,必须在使用前先检查内存分配是否成功。

说明　在使用内存之前检查指针是否为 NULL。如果指针 p 是函数的参数,那么在函数的入口处用 assert(p! = NULL)进行检查。如果是用 malloc 或 new 来申请内存,则应该用 if(p == NULL)或 if(p! = NULL)进行防错处理。

【规则 1 -5 -12】检验内存分配成功后,在使用前还必须先初始化。

说明　一般误以为内存的缺省初值全为零,故导致引用初值错误(例如数组)。所以无论用何种方式创建数组,都必须进行初始化工作,即便是赋零值也不可省略。

【规则 1 -5 -13】释放了某一块内存后,绝不可继续使用它。

说明 这种错误发生于以下3种情况：

① 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存。此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。

② 函数的return语句写错了。注意不要返回指向“栈内存”的指针或者引用，因为该内存在函数体结束时它被自动销毁了。

③ 使用free或delete释放了内存后，没有将指针设置为NULL，从而导致“野指针”的产生。

【规则1-5-14】用free或delete释放了内存之后，应立即将内存指针设置为NULL。

5.2 指针使用规则

【规则1-5-15】指针类型变量必须初始化为NULL。

【规则1-5-16】不要对指针进行复杂的逻辑或算术操作。

说明 指针加一的偏移量，通常是由指针的类型来确定的，如果进行复杂的逻辑或算术操作，则指针的位置就很难确定。

【规则1-5-17】如果指针类型明确不会改变，则应该将其设置为const类型的指针，以加强编译器的检查。

说明 这样可以防止不必要的类型转换错误。

【规则1-5-18】减少指针和数据类型的强制类型转化。

【规则1-5-19】移位操作一定要确定类型。

说明 字节类型数据在移位后还是字节类型，如将4个字节拼成一个长字，则应先把字节类型转化成长字类型。

反例

```
unsigned char ucMove = 0xA3;
unsigned long lMove;
lMove = (ucMove << 8) | ucMove;          /* 用4个字节拼成一个长字 */
lMove = (lMove << 16) | lMove;
```

正例

```
unsigned char ucMove;
unsigned long lMove;
unsigned long lTemp;
```

```
ucMove = 0xA3;
lTemp = (unsigned long) ucMove;
lMove = (lTemp << 8) | lTemp;                    /* 用4个字节拼成一个长字 */
lMove = (lMove << 16) | lMove;
```

【规则 1 -5 -20】定义公共指针的同时应对其进行初始化。

【规则 1 -5 -21】C/C ++ 程序中,指针和数组在很多情况下都可以相互替换,但实际上两者并不是等价的。

说明　数组要么在静态存储区被创建(如全局数组),要么在栈上被创建。数组名对应着(而不是指向)一块内存,其地址与容量在生命期内保持不变,只有数组的内容可以改变。而指针可以随时指向任意类型的内存块,它的特征是“可变”,所以常被用来操作动态内存。指针远比数组灵活,但也更易出错。

【规则 1 -5 -22】如果函数的参数是一个指针,绝不要用该指针去申请动态内存。

反例　下例中,Test 函数的语句 GetMemory(str, 100)并没有使 str 获得期望的内存,str 依旧是 NULL。

```
void GetMemory(char * p, int num)
{
    p = (char *)malloc(sizeof(char) * num);
}
void Test(void)
{
    char * str = NULL;
    GetMemory(str, 100); // str 仍然为 NULL
    strcpy(str, "hello"); // 运行错误
}
```

问题出在函数 GetMemory 中。编译器总是要为函数的每个参数制作临时副本,指针参数 p 的副本是_p,编译器使_p = p。如果函数体内的程序修改了_p 的内容,也将导致参数 p 的内容作相应的修改。这就是指针可以用做输出参数的原因。在本例中,_p 申请了新的内存,只是把_p 所指的内存地址改变了,但是 p 丝毫未变。所以函数 GetMemory 并不能输出任何信息。事实上,每执行一次 GetMemory 就会泄露一块内存,因为没有用 free 来释放内存。

正例　可以用函数返回值来传递动态内存,这种方法更加简单。

```
char * GetMemory(int num)
{
```

```
    char *p = (char *)malloc(sizeof(char) * num);
    return p;
}
void Test(void)
{
    char *str = NULL;
    str = GetMemory(100);
    strcpy(str, "hello");
    cout << str << endl;
    free(str);
}
```

【规则1-5-23】即使用 free 和 delete 释放指针，仍需要将指针置成 NULL。

说明 free 和 delete 只是把指针所指的内存释放掉，但并没有把指针本身删除。用调试器跟踪可发现指针经 free 操作以后其地址仍然不变(非 NULL)，只是该地址对应的内存是垃圾，指针成了“野指针”。如果此时不把指针设置为 NULL，会让人误以为指针是个合法的指针。

反例 下例中，如果程序比较长，有时记不住 p 所指的内存是否已经被释放，在继续使用 p 之前，通常会用语句 if(p! = NULL)进行防错处理。遗憾的是，此时 if 语句并不能起到防错作用，因为即便 p 不是 NULL 指针，它也可能并不是指向合法内存块的指针。

```
char *p = (char *) malloc(100);
strcpy(p, "hello");
free(p);                    // p 所指的内存被释放，但是 p 所指的地址仍然不变
...
if(p != NULL)               // 没有起到防错作用
{
strcpy(p, "world");         // 出错
}
```

【规则1-5-24】动态内存不会被自动释放。

说明 函数体内的局部变量在函数结束时会自动消亡，很多人误以为是正确的，其实不然。

反例 下例中，p 是局部指针变量，它消亡时，其所指的动态内存并不会被自动释放。

```
void Func(void)
{
    char *p = (char *) malloc(100); // 动态内存会自动释放吗?
}
```

【规则 1 -5 -25】在申请动态内存时,必须有相关代码来处理无足够大内存的情况。

说明　如果在申请动态内存时找不到足够大的内存块,malloc 和 new 将返回 NULL 指针,宣告内存申请失败。处理“内存耗尽”问题通常有以下 3 种方式。

① 判断指针是否为 NULL,如果是则立即用 return 语句终止本函数。

```
void Func(void)
{
    A * a = new A;
    if(a == NULL)
    {
        return;
    }
    ...
}
```

② 判断指针是否为 NULL,如果是则立即用 exit(1)终止整个程序的运行。

```
void Func(void)
{
    A * a = new A;
    if(a == NULL)
    {
        cout << "Memory Exhausted" << endl;
        exit(1);    //   exit(1) 表示异常返回;exit(0)表示正常返回
    }
    ...
}
```

③ 为 new 和 malloc 设置异常处理函数。例如,Visual C ++ 可以用_set_new_hander 函数为 new 设置用户自己定义的异常处理函数,也可以让 malloc 具备与 new 相同的异常处理函数。

【规则 1 -5 -26】减少没必要的指针使用,特别是较复杂的指针,如指针的指针、数组的指针、指针的数组、函数的指针等。

说明　用指针虽然灵活,但也对程序的稳定性造成一定威胁,主要原因是当要操作一个指针时,此指针可能正指向一个非法的地址。安全地使用一个指针并不是一件容易的事情。

第6章　类和类函数

【规则1-6-1】类的命名和排版应遵循一定格式。

说明　类的版式主要有两种方式：

① 将 private 类型的数据写在前面，而将 public 类型的函数写在后面。采用这种格式的程序员主张类的设计“以数据为中心”，重点关注类的内部结构。

```
class A
{
    private:
        int i, j;
        float x, y;
        ...
    public:
        void Func1(void);
        void Func2(void);
        ...
};
```

② 将 public 类型的函数写在前面，而将 private 类型的数据写在后面。采用这种格式的程序员主张类的设计“以行为为中心”，重点关注的是类应该提供的接口(或服务)。

```
class A
{
    public:
        void Func1(void);
        void Func2(void);
        ...
    private:
        int i, j;
        float x, y;
        ...
};
```

【规则1-6-2】类中的属性应声明为 private，用公有的函数访问。

说明　这样可以防止对类属性的误操作。

正例

```
class CCount
{
public:
    CCount (void);
    ~ CCount (void);
    int GetCount(void);
    void SetCount(int iCount);
private:
    int m_iCount;
}
```

【规则 1-6-3】在编写派生类的赋值函数时，注意不要忘记对基类的成员变量重新赋值。

说明　除非在派生类中调用基类的赋值函数，否则基类变量不会自动被赋值。

正例

```
class CBase
{
public:
    ...
    CBase & operate = (const CBase &other); // 类 CBase 的赋值函数
private:
    int m_iLength;
    int m_iWidth;
    int m_iHeigth;
};

class CDerived : public CBase
{
public:
    ...
    CDerived & operate = (const CDerived &other); // 类 CDerived 的赋值函数
private:
    int m_iLength;
    int m_iWidth;
    int m_iHeigth;
};
```

```
CDerived & CDerived::operate = (const CDerived &other)
{
    if (this == &other)              //检查自赋值
    {
       return *this;
    }

    CBase::operate =(other);     //对基类的数据成员重新赋值
                                 // 因为不能直接操作私有数据成员
                                 //对派生类的数据成员赋值
    m_iLength = other.m_iLength;
    m_iWidth = other.m_iWidth;
    m_iHeigth = other.m_iHeigth;

    return *this;                //返回本对象的引用
}
```

【规则1-6-4】不要在栈中分配类的实例,也不要生成全局类实例。

说明 这里所说的类,是带有构造函数的类。在栈中分配类的实例,类的构造函数和析构函数会带来很多麻烦。而全局类实例使得用户不能对该实例进行管理。

反例

```
void MemmoryFunction(…)
{
    CMyClass OneClass;       // 在栈分配类的实例可能导致构造函数的失败
    OneClass.Param1 = 2;  // 如果分配不成功,则对实例成员的访问是违规的
    …
}     // 在函数返回前,要调用类的析构函数,则又造成析构异常
```

正例

```
void MemmoryFunction(…)
{
    CMyClass *pMyClass = NULL;
    pMyClass = new CMyClass(void); // 动态申请内存

    if (pMyClass == NULL)        // 对申请的指针作有效性检查
    {
     …
     delete pMyClass ;           // 内存使用完后应释放
```

```
        pMyClass = NULL;
        …
    }
    …
}
```

【规则1-6-5】构造函数应完成简单、有效的功能，不应进行复杂的运算和大量的内存管理。

说明　如果该类有相当多的初始化工作，应生成专门的 Init(　)函数。不能在构造函数中完成全部的初始化工作，因为构造函数没有返回值，不能确定初始化是否成功。

【规则1-6-6】正确处理拷贝构造函数与赋值函数。

说明　由于并非所有的对象都会使用拷贝构造函数和赋值函数，程序员可能对这两个函数有些轻视。如果不主动编写拷贝构造函数和赋值函数，编译器将以“位拷贝”的方式自动生成缺省的函数。倘若类中含有指针变量，那么这两个缺省的函数就隐含了错误。

反例

```
class CString
{
public:
    CString(const char * pStr = NULL);          // 普通构造函数
    CString(const CString &other);              // 拷贝构造函数
    ~ CString(void);                            // 析构函数
    CString & operate = (const CString &other); // 赋值函数
public:
    char * m_pData;                             // 用于保存字符串
};
CString::CString(const char * pStr)
{
if (pStr == NULL)
{
    m_pData = new char[10];
    * m_pData = "\0";
}
else
{
    int iLength;
```

```
        iLength = strlen(pStr);
        m_pData = new char[iLength + 1];
        strcpy(m_pData, pStr);
    }
}
CString::~CString(void) // CString 的析构函数
{
    delete [] pData; // 由于 pData 是内部数据类型,也可以写成 delete pData;

}
main()
{
    CString CStringA("hello");
    CString CStringB("word");
    CString CstringC = CStringA;          // 拷贝构造函数
    CStringC = CStringB;                  // 赋值函数
    CStringB.pData = CStringA.pData;      // 这将造成以下 3 个错误
    /*1. CStringB.m_pData 原有的内存没被释放,造成内存泄露
    2. CStringB.m_pData 和 CStringA.m_pData 指向同一块内存,CStringA 和 CstringB 中任何一方变动都会影响另一方
    3. 对象被析构时,m_pData 被释放了两次。应把 m_pData 改成私有数据,用赋值函数进行赋值
    */
    ...
}
```

【规则 1-6-7】每个类只有一个析构函数和一个赋值函数,但可以有多个构造函数(包含一个拷贝构造函数,其他的称为普通构造函数)。

【规则 1-6-8】构造函数与析构函数的一个特别之处是没有返回值类型,这与返回值类型为 void 的函数是不同的!

【规则 1-6-9】如果类存在继承关系,并且基类构造函数存在没有缺省值的参数,则派生类必须在其初始化表里调用基类的构造函数。

说明　构造函数有个特殊的初始化方式叫“初始化表达式表”(简称初始化表)。初始化表位于函数参数表之后,却在函数体{} 之前。这说明该表里的初始化工作发生在函数体内的任何

代码被执行之前。

正例

```
class A
{
    A(int x); // A 的构造函数
    ...
};
class B: public A
{
    B(int x, int y);// B 的构造函数
    ...
};
B::B(int x, int y):A(x) // 在初始化表里调用 A 的构造函数
{
    ...
}
```

【规则 1 -6 -10】类的 const 常量只能在初始化表里被初始化。

说明　因为它不能在函数体内用赋值的方式来初始化。

【规则 1 -6 -11】对类中非内部数据类型的数据成员而言，其初始化工作应采用初始化表方式，这样效率更高。

说明　类的数据成员的初始化可以采用初始化表或函数体内赋值两种方式，这两种方式的效率不完全相同。非内部数据类型的成员对象应当采用第一种方式初始化，以获取更高的效率。对于内部数据类型的数据成员而言，两种初始化方式的效率几乎没有区别，但后者的程序版式似乎更清晰些。

正例

```
class A
{
    A(void); // 无参数构造函数
    A(const A &other); // 拷贝构造函数
    A & operate =( const A &other); // 赋值函数
    ...
};
class B
{
```

```
public:
    B(const A &a); // B 的构造函数
private:
    A m_a; // 成员对象
};
B::B(const A &a): m_a(a)
{
    ...
}
```

【规则 1-6-12】构造从类层次的根处开始，在每一层中，首先调用基类的构造函数，然后调用成员对象的构造函数。析构则严格按照与构造相反的次序执行，该次序是惟一的，否则编译器将无法自动执行析构过程。

【规则 1-6-13】拷贝构造函数和赋值函数非常容易混淆，常导致错写、错用。

说明　拷贝构造函数是在对象被创建时调用的，而赋值函数只能被已经存在了的对象调用。

第7章　类的继承

对象(Object)是类(Class)的一个实例(Instance)。如果将对象比作房子,那么类就是房子的设计图纸。所以面向对象设计的重点是类的设计,而不是对象的设计。对于 C ++ 程序而言,设计孤立的类是比较容易的,难的是正确设计基类及其派生类。

假设 A 是基类,B 是 A 的派生类,那么 B 将继承 A 的数据和函数。例如:

```
class A
{
  public:
      void Func1(void);
      void Func2(void);
};
class B:public A
{
  public:
      void Func3(void);
      void Func4(void);
};
main()
{
    B b;
    b.Func1(); // B 从 A 继承了函数 Func1
    b.Func2(); // B 从 A 继承了函数 Func2
    b.Func3();
    b.Func4();
}
```

C ++ 的继承特性可以提高程序的可重用性。但正因为继承特性太有用、太容易用,所以才更应防止乱用继承特性。

【规则 1 -7 -1】如果类 A 和类 B 毫不相关,不可以为了使 B 的功能更多而让 B 继承 A 的功能和属性。

【规则 1 -7 -2】若在逻辑上 B 是 A 中的“一种”(a kind of),并且 A 的所有功能和属性对 B 而言都有意义,则允许 B 继承 A 的功能和属性。

反例　例如,从生物学角度讲,鸵鸟(Ostrich)是鸟(Bird)的一种,按理说类 Ostrich 应该可以

从类 Bird 派生。但是鸵鸟不能飞,那么 Ostrich::Fly 是无意义的功能。

```
class Bird
{
  public:
      virtual void Fly(void);
      ...
};
class Ostrich : public Bird
{
      ...
};
```

正例　例如,男人(Man)是人(Human)的一种,男孩(Boy)是男人的一种。那么类 Man 可以从类 Human 派生出来,类 Boy 可以从类 Man 派生出来。

```
class Human
{
    ...
};
class Man : public Human
{
    ...
};
class Boy : public Man
{
    ...
};
```

【规则 1-7-3】若在逻辑上 A 是 B 的“一部分”(a part of),则不允许 B 从 A 派生,而是要用 A 和其他类组合出 B。

反例　例如,眼(Eye)、鼻(Nose)、口(Mouth)、耳(Ear)都是头(Head)的一部分,如果允许 Head 从 Eye、Nose、Mouth、Ear 派生而成,那么 Head 将自动具有 Look、Smell、Eat、Listen 这些功能。下例十分简短并且运行正确,但是这种设计方法却是不对的。

```
//功能正确并且代码简洁,但是设计方法不对。
class Head : public Eye, public Nose, public Mouth, public Ear
{
};
```

正例　类 Head 应该由类 Eye、Nose、Mouth、Ear 组合而成,不是派生而成。

```
class Eye
```

```
{
  public:
      void Look(void);
};
class Nose
{
  public:
      void Smell(void);
};
class Mouth
{
  public:
      void Eat(void);
};
class Ear
{
  public:
      void Listen(void);
};
//虽然代码冗长,但设计正确。
// Head 由 Eye、Nose、Mouth、Ear 组合而成
class Head
{
  public:
      void Look(void) { m_eye.Look(); }
      void Smell(void) { m_nose.Smell(); }
      void Eat(void) { m_mouth.Eat(); }
      void Listen(void) { m_ear.Listen(); }
  private:
      Eye m_eye;
      Nose m_nose;
      Mouth m_mouth;
      Ear m_ear;
};
```

【规则 1-7-4】若基类的构造函数含有不带缺省值的参数，则派生类的构造函数必须在其初始化表里调用基类的构造函数，否则编译不通过；若基类的构造函数没有参数或者所带的参数都有缺省值，则派生类的构造函数在其初始化表不需调用基类的构造函数。

正例

```
#include <iostream.h>
class Base1
{
 public:
        Base1()              { b1 = 6; }              // 无参构造函数
        Base1(int i)         { b1 = i; }              // 构造函数重载
        int get1()           { return b1; }
 private:
        int b1;
};

class Base2
{
 public:
        Base2(int i)         { b2 = i; }              // 有参构造函数,并且没有缺省值
        int get2()           { return b2; }
 private:
     int b2;
};

class Derived : public Base1, public Base2
{
     int d1;
     Base1  ob1;                                     // 成员对象
     Base2  ob2;                                     // 成员对象
public:
     // 派生类构造函数:初始化表
     // 包含成员对象 ob1 和 ob2 的初始化表及基类 Base2 的构造函数的初始化表
     // 其中 ob1 的初始化表可以没有,因为基类 Base1 有一个无参的构造函数
     // 其他 ob2 的初始化表及基类 Base2 的构造函数的初始化表不可以省略
     Derived(int x, int y, int z) : ob1(x), Base2(y), ob2(y)
      {

      .
```

```
            d1 = z;
        }
    void print( )
    {
        cout  << " \nb1 = " << get1( ) << " , ob1. b1 = " << ob1. get1( )
            << " \nb2 = " << get2( ) << " , ob2. b2 = " << ob2. get2( )
            << " \nd1 = " << d1;
        }
};

void main( )
{
    Derived d(9, 8, 7);                    // 定义派生类的对象 d,并初始化
    d. print( );
}

/*   程序的执行结果如下:
b1 = 6, ob1. b1 = 9
b2 = 8, ob2. b2 = 8
d1 = 7                                  */
```

【规则 1 - 7 - 5】基类与派生类的析构函数应该为虚(即加 virtual 关键字)。

正例

```
#include  < iostream. h >
class Base
{
  public:
      virtual  ~ Base( ) { cout << " ~ Base"  << endl ; }
};
class Derived : public Base
{
  public:
      virtual  ~ Derived( ) { cout << " ~ Derived"  << endl ; }
};
void main( void)
{
    Base * pB = new Derived; // upcast
```

```
    delete pB;
}
```

上面程序代码的输出结果为：

~ Derived

~ Base

如果析构函数不为虚，那么输出结果为：

~ Base

【规则 1-7-6】在编写派生类的赋值函数时，注意不要忘记对基类的数据成员重新赋值。

正例

```
class Base
{
  public:
      ...
      Base & operate = (const Base &other); // 类 Base 的赋值函数
  private:
      int m_i, m_j, m_k;
};
class Derived : public Base
{
  public:
      ...
      Derived & operate = (const Derived &other); // 类 Derived 的赋值函数
  private:
      int m_x, m_y, m_z;
};
Derived & Derived::operate = (const Derived &other)
{
    // 检查自赋值
    if(this == &other)
        return * this;

    //对基类的数据成员重新赋值
    Base::operate = (other); // 因为不能直接操作私有数据成员

    //对派生类的数据成员赋值
    m_x = other.m_x;
```

```
    m_y = other.m_y;
    m_z = other.m_z;

    //返回本对象的引用
    return *this;
}
```

第8章　可测试性

在设计阶段就必须考虑所编写代码的可测试性，只有提供足够的测试手段才能全面、高效地发现和解决代码中的各类问题。编写的代码是否可测试，是衡量代码质量的最基本的、最重要的尺度之一。

程序设计过程中（或程序编码完毕后），必须编写软件模块测试文档，测试文档的编写规范参见后续规范，它主要应包括设计思路、程序输入、程序输出和数据结构等部分。测试是设计的一部分。

【规则1-8-1】在同一软件开发项目组或产品组内，为准备集成测试和系统联调，要有一套统一的调试接口及相应信息输出函数，并且要有详细的说明。统一的调试接口和输出函数由模块设计和测试人员根据项目特性统一制定，由项目系统人员统一纳入系统设计中。

说明　本规则适用于以项目组或产品组形式开发的大型软件。

【规则1-8-2】在同一个软件开发项目组或产品组内，调试、测试打印出的信息应有统一的格式，并包含所在的模块名（或源文件名）及行号等信息。

说明　统一的信息格式便于集成测试。

【规则1-8-3】在编写代码之前，应预先设计好程序调试与测试的方法和手段，并设计好各种调测开关及相应测试代码（如打印函数等）。

说明　程序的调试与测试是软件生存周期中非常重要的一个阶段，如何对软件进行较全面、高效率的测试并尽可能地找出软件中的错误就成为非常关键的问题。因此在编写源代码之前，除了要有一套比较完善的测试计划外，还应设计出一系列测试代码，为单元测试、集成测试及系统联调提供方便。

【规则1-8-4】在同一软件开发项目组或产品组内，可以统一由模块设计和测试人员开发调试信息接收平台，统一对软件调试信息进行分析。

说明　本规则适用于以项目组或产品组形式开发的大型软件。

【规则1-8-5】设计人员在编程的同时要完成调试信息输出接口函数，但是测试点的选择可以由模块测试人员根据需要合理选择，测试点的选择可以根据测试用例而定，不同的测试用例应选择不同的测试点。

说明　这是为模块测试做准备。

【规则1-8-6】调测开关应分为不同级别和类型。

说明　调测开关的设置及分类应从以下几方面考虑:针对模块或系统某部分代码的调测;针对模块或系统某功能的调测;出于某种其他目的,如对性能、容量等的测试。这样做便于软件功能的调测,并且便于模块的单元测试、系统联调等。

【规则 1 -8 -7】在进行集成测试和系统联调之前,要构造好测试环境、测试项目及测试用例,同时仔细分析并优化测试用例,以提高测试效率。

说明　好的测试用例应尽可能模拟出程序所遇到的边界值、各种复杂环境及一些极端情况等。

【规则 1 -8 -8】程序的编译开关应该设置为最高优先级,编译选项设置为不优化。

说明　将编译开关置为最高优先级,可以将程序的错误尽量暴露在编译阶段,便于修正程序;将编译选项设置为不优化,是为了避免编译器优化时出错,导致程序运行出错,也是为了在程序出错时能更容易地对错误进行定位。

【规则 1 -8 -9】编程的同时要为单元测试选择恰当的测试点,并仔细构造测试代码、测试用例,同时给出明确的注释说明。测试代码部分应作为(模块中的)一个子模块,以方便测试代码在模块中的安装与拆卸(通过调测开关)。

说明　这是为单元测试做准备。

【规则 1 -8 -10】使用断言来发现软件问题,可提高代码的可测试性。

说明　断言是对某种假设条件进行检查(可理解为若条件成立则无动作,否则应报告),它可以快速发现并定位软件问题,同时对系统错误进行自动报警。断言可以对在系统中隐藏很深、用其他手段极难发现的问题进行定位,从而缩短软件问题定位时间,提高系统的可测试性。实际应用时,可根据具体情况灵活地设计断言。

正例　下面是用宏来设计的 C 语言中的一个断言,其中 NULL 为 0L。

```
#ifdef _EXAM_ASSERT_TEST_ // 若使用断言测试

void exam_assert( char * file_name, unsigned int line_no )
{
    printf( "\n[EXAM]Assert failed: %s, line %u\n",
            file_name, line_no );
    abort( );
}

#define EXAM_ASSERT( condition )
    if (condition) // 若条件成立,则无动作
            NULL;
    else // 否则报告
```

```
        exam_assert( _FILE_, _LINE_)

#else // 若不使用断言测试

#define EXAM_ASSERT( condition) NULL

#endif / *  end of ASSERT  * /
```

【规则 1 -8 -11】可用断言来检查程序正常运行时不应发生但在调测时有可能发生的非法情况。

【规则 1 -8 -12】不能用断言来检查最终产品中必须处理的错误情况。

说明　断言是用来处理不应该发生的错误情况的,对于可能会发生的且必须处理的情况要写防错程序,而不是断言。如某模块收到其他模块或链路上的消息后,要对消息的合理性进行检查,此过程为正常的错误检查,不能用断言来实现。

【规则 1 -8 -13】对较复杂的断言应加上明确的注释。

说明　为复杂的断言加注释,可澄清断言含义并减少不必要的误用。

【规则 1 -8 -14】可用断言检查函数的参数。

正例　假设某函数参数中有一个指针,那么使用指针前可用断言对它进行检查。

```
int exam_fun( unsigned char  * str )
{
    EXAM_ASSERT( str ! = NULL ); // 用断言检查“假设指针不为空”这个条件

    …//other program code
}
```

【规则 1 -8 -15】用断言能保证程序中未使用没有定义的特性或功能。

说明　假设某通信模块的消息处理接口被设计为可处理“同步消息”和“异步消息”。但在程序代码实现的当前的版本中,该消息处理接口仅实现了处理“异步消息”,且在此版本的正式发行版中,用户层(上层模块)不应产生发送“同步消息”的请求,那么在测试时可用断言检查用户是否发送了“同步消息”。

正例

```
const CHAR ASYN_EVENT  = 0;
const CHAR SYN_EVENT  = 1;
```

```
WORD MsgProcess( T_ExamMessage * ptMsg )
{
  CHAR cType;                          // 消息类型

  Assert (ptMsg != NULL);              // 用断言检查消息是否为空
  cType = GetMsgType (ptMsg);
  Assert (cType != SYN_EVENT);    // 用断言检查是否是同步消息

  ...
}
```

【规则 1 -8 -16】用断言对程序开发环境(如操作系统、编译器及硬件等)的假设进行检查。

说明　程序运行时所需的软、硬件环境及配置要求,不能用断言来检查,而必须由一段专门代码处理。用断言仅可对程序开发环境中的假设及所配置的某版本软、硬件是否具有某种功能的假设进行检查。如某网卡是否在系统运行环境中配置了,应由程序中正式代码来检查;而此网卡是否具有某设想的功能,则可由断言来检查。

对编译器提供的功能及特性假设可用断言检查,原因是软件最终产品(即运行代码或机器码)与编译器已没有任何直接关系,即软件运行过程中(注意不是编译过程中)不会也不应该对编译器的功能提出任何需求。

正例

```
// 用断言来检查编译器的 int 型数据占用的内存空间是否为 2
EXAM_ASSERT( sizeof( int ) == 2 );
```

【规则 1 -8 -17】正式软件产品中应把断言及其他调测代码去掉(即把有关的调测开关关掉)。

说明　这样可加快软件运行速度。

【规则 1 -8 -18】在软件系统中设置与取消有关测试手段时,应保证不会对软件实现的功能等产生影响。

说明　即有测试代码的软件和关掉测试代码的软件,在功能、行为上应一致。

【规则 1 -8 -19】用调测开关来切换软件的 DEBUG 版和正式版,而不要同时存在正式版本和 DEBUG 版本的不同源文件,以减少维护的难度。

【规则 1 -8 -20】软件的 DEBUG 版本和发行版本应该统一维护,不允许分开处理,并且要时刻注意保证两个版本在实现功能上的一致性。

【规则 1 -8 -21】去除代码运行的随机性(如去掉无用的数据、代码及尽可能防止函数中的互相调用等),让函数运行的结果可预测,并使出现的错误可再现。

【规则 1 -8 -22】编写防错程序,然后在处理错误之后可用断言报告错误。

正例　假如某模块收到通信链路上的消息,则应对消息的合法性进行检查,若消息类别不是通信协议中规定的,则应进行出错处理,之后可用断言报告。

```
#ifdef _EXAM_ASSERT_TEST_ // 若使用断言测试

/* Notice: this function does not call 'abort' to exit program */
void assert_report( char * file_name, unsigned int line_no )
{
    printf( "\n[EXAM]Error Report: %s, line %u\n",
                file_name, line_no );
}

#define ASSERT_REPORT( condition )
    if ( condition)     // 若条件成立,则无动作
        NULL;
    else     // 否则报告
        assert_report ( _FILE_,_LINE_ )
#else  // 若不使用断言测试
#define ASSERT_REPORT( condition ) NULL
#endif /* end of ASSERT */

int msg_handle( unsigned char msg_name, unsigned char * msg )
{
    switch( msg_name )
    {
        case MSG_ONE:
        …// 消息 MSG_ONE 处理
        return MSG_HANDLE_SUCCESS;

        …// 其他合法消息处理

    default:
        …// 消息出错处理
        ASSERT_REPORT( FALSE ); // “合法”消息不成立,报告
```

```
                return MSG_HANDLE_ERROR;
            }
        }
```

【规则 1 -8 -23】在设计时应考虑以下常用的发现错误的方法。

说明　以下发现错误的方法可以为编写可测试性代码提供思路：

- 使用所有数据建立假设
- 求精发现错误的测试用例
- 通过不同的方法再生错误
- 产生更多的数据以生成更多的假设
- 提出尽可能多的假设
- 缩小可疑代码区
- 检查最近作过修改的代码
- 扩展可疑代码区
- 逐步集成
- 重点检查以前出过错的子程序
- 耐心检查
- 检查一般错误
- 使用交互式调试法

【规则 1 -8 -24】在设计时考虑以下常见改正错误的方法。

说明　以下改正错误的方法可以为编写可测试性代码提供思路：

- 理解问题的实质
- 理解整个程序
- 确诊错误
- 保存初始源代码
- 仅为某种原因修改代码
- 一次作一个修改
- 验证修改
- 寻找相似错误

【规则 1 -8 -25】程序开发人员对自己模块内的函数必须通过有效的方法进行测试，保证所有代码都执行到。

【规则 1 -8 -26】单元测试要求至少达到语句覆盖。

【规则 1 -8 -27】单元测试应跟踪每一条语句，并观察数据流及变量的变化。

【规则 1 -8 -28】清理、整理或优化后的代码要经过审查及测试。

【规则 1 -8 -29】代码版本升级要经过严格测试。

【规则 1 -8 -30】使用工具软件对代码版本进行维护。

【规则 1 -8 -31】对正式版本软件的任何修改都应有详细的文档记录。

【规则 1 -8 -32】发现错误立即修改，并且要记录下来。

【规则 1 -8 -33】应在汇编级跟踪关键的代码。

【规则 1 -8 -34】仔细设计并分析测试用例，使测试用例覆盖尽可能多的情况，以提高测试用例的效率。

【规则 1 -8 -35】尽可能模拟出程序的各种出错情况，对出错处理代码进行充分的测试。

【规则 1 -8 -36】仔细测试代码处理数据、变量的边界情况。

【规则 1 -8 -37】保留测试信息，以便分析、总结经验及进行更充分的测试。

【规则 1 -8 -38】不应通过“试”来解决问题，应寻找问题的根本原因。

【规则 1 -8 -39】对自动消失的错误进行分析，搞清楚错误是如何消失的。

【规则 1 -8 -40】修改错误不仅要治表，更要治本。

【规则 1 -8 -41】测试时应设法使很少发生的事件经常发生。

【规则 1 -8 -42】坚持在编码阶段就对代码进行彻底的单元测试，不要等以后的测试工作来发现问题。

第9章　程序效率和质量保证

9.1　程序效率

【规则1-9-1】编程时要经常注意代码的效率。

说明　代码效率分为全局效率、局部效率、时间效率及空间效率。全局效率是站在整个系统的角度上的系统效率；局部效率是站在模块或函数角度上的效率；时间效率是程序处理输入任务所需的时间长短；空间效率是程序所需的内存空间，如机器代码空间大小、数据空间大小、栈空间大小等。

【规则1-9-2】在保证软件系统的正确性、稳定性、可读性及可测试性的前提下，提高代码效率。

说明　不能一味地追求代码效率，而对软件的正确性、稳定性、可读性及可测试性造成影响。

【规则1-9-3】局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。

【规则1-9-4】通过优化数据结构及程序算法来提高空间效率。

说明　这种方式是解决软件空间效率的根本办法。

反例　如下记录学生学习成绩的结构不合理。

```
typedef unsigned char BYTE;
typedef unsigned short WORD;

typedef struct STUDENT_SCORE_STRU
{
    BYTE name[8];
    BYTE age;
    BYTE sex;
    BYTE class;
    BYTE subject;
    float score;
} STUDENT_SCORE;
```

正例　因为每位学生都有多科学习成绩，故如上结构将占用较大空间。改进后，即分为两个

结构,总的存贮空间将变小,操作也变得更方便。

```
typedef struct STUDENT_STRU
{
    BYTE name[8];
    BYTE age;
    BYTE sex;
    BYTE class;
} STUDENT;

typedef struct STUDENT_SCORE_STRU
{
    WORD student_index;
    BYTE subject;
    float score;
} STUDENT_SCORE;
```

【规则1-9-5】循环体内工作量最小化。

说明 应仔细考虑循环体内的语句是否可以放在循环体之外,使循环体内工作量尽可能减少,从而提高程序的时间效率。

反例 如下代码效率不高。

```
for (ind = 0; ind < MAX_ADD_NUMBER; ind ++)
{
    sum += ind;
    back_sum = sum; /* backup sum */
}
```

正例 语句"back_sum = sum;"完全可以放在for语句之后。

```
for (ind = 0; ind < MAX_ADD_NUMBER; ind ++)
{
    sum += ind;
}
back_sum = sum; /* backup sum */
```

【规则1-9-6】仔细考查、分析系统及模块处理输入(如事务、消息等)的方式,并加以改进。

【规则1-9-7】对模块中函数的划分及组织方式进行分析、优化,改进模块中函数的组织结构,提高程序效率。

说明　软件系统的效率主要与算法、处理任务方式、系统功能及函数结构有很大关系，仅在代码上下功夫一般不能解决根本问题。

【规则 1 -9 -8】编程时，要随时留心代码效率；优化代码时，要考虑周全。

【规则 1 -9 -9】不应花过多的时间去提高调用不很频繁的函数代码效率。

说明　对代码优化可提高效率，但若考虑不周很有可能引起严重后果。

【规则 1 -9 -10】要仔细地构造或直接用汇编编写调用频繁或性能要求极高的函数。

说明　只有对编译系统产生机器码的方式以及硬件系统较为熟悉时，才可使用汇编嵌入方式。嵌入汇编可提高时间及空间效率，但也存在一定风险。

【规则 1 -9 -11】在保证程序质量的前提下，通过压缩代码量、去掉不必要代码以及减少不必要的局部变量和全局变量，可提高空间效率。

说明　这种方式对提高空间效率可起到一定作用，但往往不能解决根本问题。

【规则 1 -9 -12】尽量减少循环嵌套层次。

【规则 1 -9 -13】避免循环体内出现判断语句，应将循环语句置于判断语句的代码块之中。

说明　这样做的目的是减少判断次数。循环体中的判断语句是否可以移到循环体外，要视程序的具体情况而定。一般情况下，与循环变量无关的判断语句可以移到循环体外，而相关的则不可以。

反例　如下代码效率稍低。

```
for (ind = 0; ind < MAX_RECT_NUMBER; ind ++)
{
    if (data_type == RECT_AREA)
    {
        area_sum += rect_area[ind];
    }
    else
    {
        rect_length_sum += rect[ind].length;
        rect_width_sum += rect[ind].width;
    }
}
```

正例　因为判断语句与循环变量无关，故可做如下改进，以减少判断次数。

```
if (data_type == RECT_AREA)
{
```

```
        for (ind = 0; ind < MAX_RECT_NUMBER; ind ++)
        {
            area_sum += rect_area[ind];
        }
    }
    else
    {
        for (ind = 0; ind < MAX_RECT_NUMBER; ind ++)
        {
            rect_length_sum += rect[ind].length;
            rect_width_sum += rect[ind].width;
        }
    }
```

【规则1-9-14】尽量用乘法或其他方法代替除法,特别是浮点运算中的除法。

说明 浮点运算除法要占用较多的CPU资源。

反例 如下表达式运算可能要占较多的CPU资源。

```
#define PAI 3.1416
radius = circle_length / (2 * PAI);
```

正例 应把上面的浮点除法改为浮点乘法。

```
#define PAI_RECIPROCAL (1 / 3.1416 ) // 编译器编译时,将生成具体浮点数
radius = circle_length * PAI_RECIPROCAL / 2;
```

【规则1-9-15】不要一味追求紧凑的代码。

说明 因为紧凑的代码并不代表高效的机器码。

【规则1-9-16】在优化程序的效率时,应当先找出限制效率的“瓶颈”,不要在无关紧要之处优化。

【规则1-9-17】应先优化数据结构和算法,再优化执行代码。

【规则1-9-18】有时候时间效率和空间效率可能对立,此时应当分析哪个更重要,作出适当的折中。例如多花费一些内存来提高性能。

9.2　质量保证

【规则 1 -9 -19】严格遵守代码质量保证优先原则。

说明　代码质量保证优先原则包括：

① 正确性：指程序要实现设计要求的功能。

② 稳定性、安全性：指程序稳定、可靠、安全。

③ 可测试性：指程序要具有良好的可测试性。

④ 规范/可读性，指程序书写风格、命名规则等要符合规范。

⑤ 全局效率：指软件系统的整体效率。

⑥ 局部效率：指某个模块/子模块/函数本身的效率。

⑦ 个人表达方式/个人方便性：指个人编程习惯。

【规则 1 -9 -20】只引用属于自己的存贮空间。

说明　若模块封装得较好，那么一般不会发生非法引用其他存储空间的问题。

【规则 1 -9 -21】防止引用已经释放的内存空间。

说明　在实际编程过程中，稍不留心就会出现在一个模块中释放了某个内存块(如 C 语言指针)，而另一模块在随后的某个时刻又使用了它。要防止这种情况的发生。

【规则 1 -9 -22】在过程/函数中分配的内存，应在过程/函数退出之前释放。

【规则 1 -9 -23】在过程/函数中申请的(为打开文件而使用的)文件句柄，应在过程/函数退出之前关闭。

说明　分配的内存不释放或文件句柄不关闭，是较常见的错误，而且稍不注意就有可能发生。这类错误往往会引起很严重的后果，且难以定位。

反例　以下函数在退出之前，没有把分配的内存释放。

```
typedef unsigned char BYTE;
int example_fun( BYTE gt_len, BYTE *gt_code )
{
    BYTE *gt_buf;

    gt_buf = (BYTE *) malloc (MAX_GT_LENGTH);
    ... //program code, include check gt_buf if or not NULL.
    /* global title length error */
    if (gt_len > MAX_GT_LENGTH)
```

```
        {
            return GT_LENGTH_ERROR; // 忘了释放 gt_buf
        }

        … // other program code
    }
```

正例

```
int example_fun( BYTE gt_len, BYTE * gt_code )
{
    BYTE * gt_buf;

    gt_buf = (BYTE * ) malloc ( MAX_GT_LENGTH );
    … // program code, include check gt_buf if or not NULL.

    /* global title length error */
    if (gt_len > MAX_GT_LENGTH)
    {
        free( gt_buf ); // 退出之前释放 gt_buf
        return GT_LENGTH_ERROR;
    }

    … // other program code
}
```

【规则 1 -9 -24】认真处理程序中可能的各种出错情况。

【规则 1 -9 -25】系统运行之初,要对加载到系统中的数据进行一致性检查。

说明　使用不一致的数据,容易使系统进入混乱状态和不可知状态。

【规则 1 -9 -26】严禁随意更改其他模块或系统的有关设置和配置。

说明　编程时,不能随心所欲地更改不属于自己模块的有关设置,如常量、数组的大小等。

【规则 1 -9 -27】不能随意改变与其他模块的接口。

【规则 1 -9 -28】编程时,要防止差 1 的错误。

说明　此类错误一般是由于把“ <= ”误写成“ < ”或把“ >= ”误写成“ > ”等造成的,由此引

起的后果,很多情况下是很严重的,所以编程时,一定要特别小心。当编完程序后,应对这些操作符进行彻底检查。

【规则 1 -9 -29】要时刻注意易混淆的操作符。当编完程序后,应从头至尾检查一遍这些操作符,以防止拼写错误。

说明　形式相近的操作符最容易引起误用,如 C/C ++ 中的“ = ”与“ == ”、“ | ”与“ || ”、“ & ”与“ && ”等,若拼写错了,编译器不一定能够检查出来。

反例 1　如把“ & ”写成“ && ”。

```
ret_flg = (pmsg -> ret_flg && RETURN_MASK);
```

正例 1

```
ret_flg = (pmsg -> ret_flg & RETURN_MASK);
```

反例 2　把“ && ”写成“ & ”。

```
rpt_flg = (VALID_TASK_NO( taskno ) & DATA_NOT_ZERO( stat_data ));
```

正例 2

```
rpt_flg = (VALID_TASK_NO( taskno ) && DATA_NOT_ZERO( stat_data ));
```

【规则 1 -9 -30】在 UNIX 下,多线程中的子线程退出时,必须采用主动退出方式,即子线程应提供 return 出口。

【规则 1 -9 -31】不使用与硬件或操作系统关系很大的语句,而使用建议的标准语句,以提高软件的可移植性和可重用性。

【规则 1 -9 -32】除非为了满足特殊需求,应避免使用嵌入式汇编,以提高软件的可移植性。

说明　在程序中使用嵌入式汇编,一般会对软件的可移植性产生较大的影响。

【规则 1 -9 -33】精心地构造、划分子模块,并按“接口”部分及“内核”部分合理地组织子模块,以提高“内核”部分的可移植性和可重用性。

说明　对不同产品中的某个功能相同的模块,若能做到其内核部分完全或基本一致,那么无论对产品的测试、维护,还是对以后产品的升级都会有很大帮助。

【规则 1 -9 -34】注意表达式是否会出现上溢或下溢错误。

反例　如下程序将造成变量下溢。

```
unsigned char size ;
while (size --> = 0) // 将出现下溢
{
    … // program code
}
```

当 size 等于 0 时，再减 1 不会小于 0，而是 0xFF，故程序是一个死循环。

正例　应做如下修改。

```
char size; // 从 unsigned char 改为 char
while (size --> = 0)
{
    … // program code
}
```

【规则 1-9-35】使用变量时要注意其边界值的情况。

反例　如 C 语言中字符型变量，有效值范围为 -128 ~ 127。故以下表达式的计算存在一定风险。

```
char chr = 127;
int sum = 200;

chr += 1; // 127 为 chr 的边界值，再加 1 将使 chr 上溢到 -128，而不是 128
sum += chr; // 故 sum 的结果不是 328，而是 72
```

若 chr 与 sum 为同一种类型，可按如下方式书写最后两条语句，可能会好些。

```
sum = sum + chr + 1;
```

【规则 1-9-36】注意程序机器码大小（如指令空间大小、数据空间大小、堆栈空间大小等）是否超出系统有关限制。

【规则 1-9-37】为用户提供良好的接口界面，使用户能较充分地了解系统内部运行状态及有关系统出错情况。

【规则 1-9-38】系统应具有一定的容错能力，对一些错误事件（如用户误操作等）能进行自动补救。

【规则 1-9-39】对一些具有危险性的操作代码（如写硬盘、删数据等）要仔细考虑，防止对数据、硬件等的安全构成危害，以提高系统的安全性。

【规则 1-9-40】当心数据类型转换时发生错误。尽量使用显式的数据类型转换，而避免让编译器进行隐式的数据类型转换。

【规则 1-9-41】避免编写技巧性很高的代码。

【规则 1 -9 -42】如果原有的代码质量比较好，尽量复用它。不要修补很差的代码，而应当重新编写。

【规则 1 -9 -43】尽量使用标准库函数，不要“发明”已经存在的库函数。

第 10 章　错误和异常处理规范

【规则 1-10-1】出错类型的定义必须要有统一的约定。

说明　① 整个系统软件产品的出错定义要一致。

② 同一模块层次的出错类型统一定义。

③ 出错类型分为错误、警告、提示等 3 类信息。

④ 错误代码统一用宏描述，并且放在一个头文件中。

⑤ 出错代码的宏定义还要加注对这个代码的说明。

⑥ 应有相应的文档指明出现代码的定义规则。

正例　在 ErrorDef.h 文件中有如下定义：

```
// *****************************************************
// Error Code Macro Define
// *****************************************************
// Error Code
#define E_OK                        0x0        // 没有错误
#define E_NO_ENOUGH_MEMORY          0x1001     // 内存不足
#define E_INVALID_HANDLE            0x1002     // 无效的句柄

// Warning Code
#define W_CUT_RECV_DATA             0x2001     // 裁剪接收数据
#define W_ FIND_OLD_MESSAGE         0x2002     // 发现老的消息

// Information Code
#define I_SEND_SUCCEED              0x9001     // 发送成功
```

【规则 1-10-2】程序中必须要编写异常捕获和处理。

说明　在程序中会出现各种异常，如除 0 错误、内存错误等，对这些异常都要进行相应的处理。

【规则 1-10-3】整个软件系统应该采用统一的断言。如果系统不提供断言，则应该自己构造一个统一的断言供编程时使用。

说明　整个软件系统提供一个统一的断言函数，如 Assert(exp)，同时可提供不同的宏进行定义(可根据具体情况灵活设计)，如：

① #define ASSERT_EXIT_M:中断当前程序执行,打印中断发生的文件、行号,该宏一般在单调时使用。

② #define ASSERT_CONTINUE_M:打印程序发生错误或异常的文件、行号,继续进行后续的操作,该宏一般在联调时使用。

③ #define ASSERT_OK_M 空操作:程序发生错误情况时,继续进行,可以通过适当的方式通知后台的监控或统计程序,该宏一般在 RELEASE 版本中使用。

【规则 1 -10 -4】使用断言捕捉不应该发生的异常情况。不要混淆异常情况与错误情况之间的区别,后者是必然存在的并且是一定要作出处理的。

【规则 1 -10 -5】指向指针的指针及多级指针必须逐级检查。

说明　对指针逐级检查,有利于准确定位错误。

反例

```
Assert (ptStru -> ptForward -> ptBackward ! = NULL);
```

正例

```
Assert ( (ptStru ! = NULL)
        && (ptStru -> ptForward ! = NULL)
        && (ptStru -> ptForward -> ptBackward ! = NULL));
```

【规则 1 -10 -6】尽可能模拟程序中各种出错状态,以测试软件对出错状态的处理。

说明　需要确定系统中可能发生哪些事情,并使它们发生以检测处理是否正常。对系统中的小概率事件,也应设法使其重现。

【规则 1 -10 -7】使用断言检查函数输入参数的有效性、合法性。

说明　检查函数的输入参数是否合法,如输入参数为指针,则可用断言检查该指针是否为空;如输入参数为索引,则检查索引是否在值域范围内。

正例

```
BYTE StoreCsrMsg(WORD wIndex, T_CMServReq * ptMsgCSR)
{
    WORD        wStoreIndex;
    T_FuncRet   tFuncRet;

    Assert (wIndex < MAX_DATA_AREA_NUM_A);      // 使用断言检查索引
    Assert (ptMsgCSR ! = NULL);                 // 使用断言检查指针

    ...                                         // 其他代码
```

```
    return OK_M;
}
```

【规则 1-10-8】对所有具有返回值的接口函数的返回结果进行断言检查。

说明　对接口函数的返回结果进行检查,可以避免程序运行过程中因使用不正确的返回值而引起的错误。

正例

```
BYTE HandleTpWaitAssEvent(T_CcuData *ptUdata, BYTE *pucMsg)
{
    T_CacAssignFail *ptAssignfail;
    T_CccData *ptCdata;

    ptAssignfail = (T_CacAssignFail *)pbMsg;

    ...                                    // 其他代码
    ptCdata = GetCallData(ptUdata->waCallindex[0]);
    Assert (ptCdata != NULL);      // 使用断言对函数的返回结果进行检查
    ...                                    // 其他代码

    return CCNO_M;
}
```

第11章　其他规范

11.1　可　读　性

【规则 1 -11 -1】注意运算符的优先级，并用括号明确表达式的操作顺序，避免使用默认的优先级。

说明　这样可防止阅读程序时产生误解，也可防止因默认的优先级与设计思想不符而导致程序出错。

【规则 1 -11 -2】避免使用不易理解的数字，用有意义的标识来替代。涉及物理状态或者物理意义的常量，不应直接使用数字，必须用有意义的枚举或宏来代替。

反例　如下的程序可读性差。

```
if (Trunk[index].trunk_state == 0)
{
    Trunk[index].trunk_state = 1;
    … // program code
}
```

正例　应改为如下形式。

```
#define TRUNK_IDLE 0
#define TRUNK_BUSY 1

if (Trunk[index].trunk_state == TRUNK_IDLE)
{
    Trunk[index].trunk_state = TRUNK_BUSY;
    … // program code
}
```

【规则 1 -11 -3】除非确实需要，否则不要使用难懂的、技巧性很高的语句。

说明　高技巧语句不等于高效率的程序，实际上程序的效率关键取决于算法。

反例　如下表达式，考虑不周就可能出问题，也较难理解。

```
* stat_poi ++ += 1;
```

```
    *  ++ stat_poi  += 1;
```

正例　应分别改为如下形式。

```
    * stat_poi  += 1;
    stat_poi ++ ;              // 这两条语句功能相当于“ * stat_poi  +++= 1; ”
    ++ stat_poi;
    * stat_poi  += 1;      // 这两条语句功能相当于“ *  ++ stat_poi  += 1; ”
```

11.2　宏

【规则 1 -11 -4】用宏定义表达式时,要使用完备的括号。

反例　如下定义的宏都存在一定的风险。

```
    #define RECTANGLE_AREA( a, b ) a * b
    #define RECTANGLE_AREA( a, b ) (a * b)
    #define RECTANGLE_AREA( a, b ) (a) * (b)
```

正例

```
    #define RECTANGLE_AREA( a, b ) ((a) * (b))
```

【规则 1 -11 -5】将宏所定义的多条表达式放在花括号中。

反例　下面的宏定义语句书写不规范。同时,for 语句的书写也不符合规范。

```
    #define INTI_RECT_VALUE( a, b )a = 0;b = 0;

    for (index = 0; index < RECT_TOTAL_NUM; index ++ )
          INTI_RECT_VALUE( rect. a, rect. b );
```

正例　正确的用法应为:

```
    #define INTI_RECT_VALUE( a, b )\
    {\
        a = 0;\
        b = 0;\
    }

    for (index = 0; index < RECT_TOTAL_NUM; index ++)
    {
        INTI_RECT_VALUE( rect[index]. a, rect[index]. b );
    }
```

【规则 1 -11 -6】预编译条件不应分离一条完整的语句。

反例

```
if (( cond == GLRUN)
    #ifdef DEBUG
      || (cond == GLWAIT)
    #endif
   )
{
}
```

正例

```
#ifdef DEBUG
    if( cond == GLRUN || cond == GLWAIT )
#else
    if( cond == GLRUN )
#endif
{
}
```

【规则 1 -11 -7】在宏定义中合并预编译条件。

反例

```
#ifdef EXPORT
    for ( i = 0; i < MAX_MSXRSM; i ++ )
#else
    for ( i = 0; i < MAX_MSRSM; i ++ )
#endif
```

正例　在头文件中,定义语句为:

```
#ifdef EXPORT
    #define MAX_MS_RSM MAX_MSXRSM
#else
    #define MAX_MS_RSM MAX_MSRSM
#endif
```

在源文件中,调用该宏的语句为:

```
for( i = 0; i < MAX_MS_RSM; i ++ )
```

11.3　代码编辑、编译、审查

【规则 1-11-8】对程序进行编译时，应打开编译器的所有警告开关。

【规则 1-11-9】在产品软件(项目)中，要统一编译开关选项的设置。

【规则 1-11-10】通过代码走读及审查方式对代码进行检查。

说明　代码走读主要是对程序的编程风格如注释、命名等以及编程时易出错的内容进行检查，可由开发人员自己或开发人员交叉的方式进行；代码审查主要是对程序实现的功能及程序的稳定性、安全性、可靠性等进行检查及评审，可通过自审、交叉审核或指定部门抽查等方式进行。

【规则 1-11-11】测试人员测试产品之前，应对代码进行抽查及评审。

【规则 1-11-12】编写代码时要注意随时保存，并定期备份，防止由于断电、硬盘损坏等原因造成代码丢失。

【规则 1-11-13】要小心地使用编辑器提供的块拷贝功能编程。

说明　当某段代码与另一段代码的处理功能相似时，许多开发人员都用编辑器提供的块拷贝功能来完成这段代码的编写。由于程序功能相近，故所使用的变量、采用的表达式等在功能及命名上可能都很相近，所以使用块拷贝时要注意，除了修改相应的程序外，一定要把使用的每个变量仔细查看一遍，以保证其正确性。不应指望编译器能查出所有这种错误，比如当使用的是全局变量时，就有可能使某种错误隐藏下来。

【规则 1-11-14】合理地设计软件系统目录，方便开发人员使用。

说明　方便、合适的软件系统目录，可提高工作效率。目录构造的原则是方便有关源程序的存储、查询、编译、链接等工作，同时目录中还应有工作目录和工具目录。所有的编译、链接等工作在工作目录中进行；有关文件编辑器、文件查找等工具可存放在工具目录中。

第二部分

Java 语言编程规范

第1章　程序组织规则

【规则2-1-1】Java程序使用的文件后缀名分别是:Java源文件(Java source)为.JAVA;Java字节码文件(Java class)为.CLASS。

【规则2-1-2】每个文件的组成部分应该以空行和适当的注释分开。

【规则2-1-3】一个文件应尽量避免超过2 000行。

【规则2-1-4】Java程序应该包括单个public的class或者interface。

【规则2-1-5】当私有的classes及interface与一个公有的class关联的时候,应该将它们放在同一个文件中。公有的class应该是文件的第一个class或interface。

【规则2-1-6】Java文件应遵循一定的组织顺序。

说明　Java文件的组织顺序一般为:

- 文件头注释
- 包(Package)与导入类声明
- 类与接口声明

【规则2-1-7】所有的程序文件应该以注释开始,该注释列出类名、版本信息、日期以及版权等。

说明

```
/*
* Classname
*
* Version information
*
* Date
*
* Copyright notice
*/
```

【规则2-1-8】多数Java源文件中,第一个非注释行是包语句。在它之后可以跟引入语句。

说明　package java. awt;

import java. awt. peer. CanvasPeer;

【规则 2 -1 -9】package 行应放在 import 行之前，import 中标准的包名应放在本地的包名之前，而且按照字母顺序排列。

说明　如果使用 import 一次导入同一个中的多个类，则应该用“ * ”来处理。

正例

```
package hotlava. net. stats;
import java. io. * ;
import java. util. Observable;
import hotlava. util. Application;
```

这里，使用“java. io. * ”来导入 Java. io 包中的所有类。

【规则 2 -1 -10】注意不要用“import java. awt. * ”方法导入类，要一个一个地将类导入，并且按照字母顺序排列。

说明　不同 package 或不同功能的 class 要以一个空行隔开，这样阅读程序的时候能够快速了解并定位到相关的 class。

正例

```
package cscweb. preserve. model;

import java. util. ArrayList;

import org. apache. log4j. Category;

import cscweb. common. ejb. CSCWebException;
import cscweb. common. ejb. PJCodeBook;
import cscweb. util. DateUtil;
import cscweb. util. MessageManager;

import cscweb. preserve. data. HozenActionDefMasterData;
import cscweb. preserve. ejb. PreserveDefReg;
import cscweb. preserve. ejb. PreserveDefRegHome;
```

【规则 2 -1 -11】在类定义前，应有类的注释。

说明　此注释一般是用来解释类的。

正例

```
/**
 * A class representing a set of packet and byte counters
 * It is observable to allow it to be watched, but only
 * reports changes when the current set is complete
 */
```

【规则 2-1-12】在类定义中,包含了在不同的行的 extends 和 implements。

正例

```
public class CounterSet
extends Observable
implements Cloneable
```

【规则 2-1-13】在类的定义后,是有关类的成员变量的定义。

说明　proceted、private 和 package 定义的成员变量,如果其名字含义明确,也可以没有注释。

正例

```
/**
 * Packet counters
 */
protected int[] packets;
```

【规则 2-1-14】对于类变量的存取方法,若其功能只是简单地用来将类的变量赋值及获取值,则该方法可以写在一行上。

正例

```
/**
 * Get the counters
 * @return an array containing the statistical data. This array has been
 * freshly allocated and can be modified by the caller.
 */
public int[] getPackets(){ return copyArray(packets, offset); }
public int[]getBytes(){ return copyArray(bytes, offset); }

public int[]getPackets(){ return packets; }
public void setPackets(int[] packets){ this.packets = packets; }
    //其他的方法不要写在一行上
```

【规则 2-1-15】对构造函数,应该用递增的方式写(比如,参数多的写在后面)。

说明　访问类型("public","private"等)和 任何"static","final"或"synchronized"应该在一

行中,并且方法和参数另写一行,这样可以使方法和参数更易读。

正例

```
public
CounterSet(int size){
  this.size = size;
}
```

【规则 2-1-16】如果这个类是可以被克隆的,那么必须要有 clone 方法。

正例

```
public
Object clone(){
  try {
    CounterSet obj = (CounterSet)super.clone();
    obj.packets = (int[])packets.clone();
    obj.size = size;
    return obj;
  }catch(CloneNotSupportedException e){
    throw new InternalError("Unexpected CloneNotSUpportedException:" +
    e.getMessage());
  }
}
```

【规则 2-1-17】类方法书写应规范。

正例

```
/**
 * Set the packet counters
 * (such as when restoring from a database)
 */
protected final
void setArray(int[] r1, int[] r2, int[] r3, int[] r4)
throws IllegalArgumentException
{

//
//Ensure the arrays are of equal size
//
if(r1.length != r2.length || r1.length != r3.length || r1.length != r4.length)
  throw new IllegalArgumentException("Arrays must be of the same size");
System.arraycopy(r1, 0, r3, 0, r1.length);
```

```
        System. arraycopy( r2, 0, r4, 0, r1. length);
    }
```

【规则 2 - 1 - 18】每一个类都应该定义一个 toString 方法。

正例

```
public
String toString( ) {
    String retval = "CounterSet:" ;
        for( int i = 0; i < data. length( ) ;i ++ ) {
            retval + = data. bytes. toString( ) ;
            retval + = data. packets. toString( ) ;
        }
        return retval;
    }
}
```

【规则 2 - 1 - 19】如果 main(String[ ])方法已经定义了,那么它应该写在类的底部。

【规则 2 - 1 - 20】每个缩进单元应是 4 个空格。

说明　不要在源文件中保存 Tab 字符。在使用不同的源代码管理工具时,Tab 字符将因为用户设置的不同而扩展为不同的宽度。

【规则 2 - 1 - 21】必须用 Javadoc 来为类生成文档。

说明　不仅因为它是标准的,而且是被各种 Java 编译器都认可的方法。

【规则 2 - 1 - 22】每行避免超过 80 个字符。

说明　因为有的终端和编辑工具不能很好地处理超过 80 个字符的行。

【规则 2 - 1 - 23】{ }中的每行语句应该单独作为一行。

反例

```
if(i > 0) {i ++ };//错误,“{”和“}”在同一行
```

正例

```
if(i > 0) {
i ++
};          //正确,“{”单独作为一行
```

【规则 2 - 1 - 24】左括号和其后一个字符之间不应该出现空格,同样,右括号和其前一个字符之间也不应该出现空格。

反例

```
CallProc(AParameter);//错误
```

正例

```
CallProc(AParameter);//正确
```

【规则 2-1-25】不要在语句中使用无意义的括号。

说明　括号只应该为达到某种目的而出现在源代码中。

反例

```
if((I) =42){          //错误,括号毫无意义
```

正例

```
if(I ==42) or(J ==42) then          //正确,的确需要括号
```

【规则 2-1-26】在一行内无法完成一个语句的编写或不合适时,应该用以下的原则进行分行:

- 在逗号之后
- 在一个操作符之前
- 在优先度高的地方

新行的开始应该与上一行同一级别处垂直对齐。如果满足以上的原则的新行的开始比较靠右,应该用 8 个空格进行缩排。

【规则 2-1-27】程序文件中两部分之间和在不同的类与接口之间应该要用两行空行分开。

【规则 2-1-28】用空行将一段逻辑相关的代码与其他代码分开可以提高代码的可读性。

下面的情况应该用一行空行:

- 两方法之间
- 在方法内部的局部变量定义与它的第一个语句之间
- 在一个块注释或单行注释之前
- 在一个方法内的不同的逻辑操作部分之间

【规则 2-1-29】关键字与括号之间应该用空格分开。

正例

```
while(true){
    ...
}
```

注意,空格不应该用在方法的名字与它后面的括号之间,这有助于区分关键字与方法的调用。

【规则 2-1-30】在参数列表中,逗号后面应该有一个空格。

说明　但参数名与逗号之间就不要用空格了。

【规则 2 -1 -31】所有的二元操作符除了“.”外都应该与操作数之间用空格分开。

说明　但在一元操作符(++ 、--)与它的操作子之间不需用空格分开。

正例

```
a + = c + d;
a = (a + b)/(c * d);

while(d ++ = s ++ ){
    n ++ ;
}
printSize( "size is" + foo + "\n" );
```

【规则 2 -1 -32】for 语句中的表达式应该用空格隔开。

正例

```
for(expr1 ;expr2 ;expr3 )
```

【规则 2 -1 -33】强制类型转换后应该有一空格。

正例

```
myMethod( (byte) aNum , (Object) x) ;
myMethod( (int) (cp +5) , ( (int) (i +3) ) +1) ;
```

第2章　命名约定

【规则2-2-1】包名前缀不应该重复，并且一律小写。

【规则2-2-2】包标识符间用点号分隔。

说明　为了使包的名字更易读，Sun 公司建议包名中的标识符用点号来分隔。

【规则2-2-3】Sun 公司的标准 Java 分配包用标识符“java.”开头。

【规则2-2-4】全局包的名字用机构的 Internet 保留域名开头。

说明　包的名字一般为一些最高级别的域名，如 com、edu、gov、mil、net、org 等，或是以两个字母缩写的国家代号，或者一个项目的名称。接下来的名字可以用公司的产品名、功能块名、组织名、机器名等来命名。

正例

```
com. sun. eng
com. apple. quicktime. v2
edu. cmu. cs. bovik. cheese
```

【规则2-2-5】包的命名全都是小写字母，即便中间的单词亦是如此。

说明　对于域名扩展名称，如 com、org、net 或者 edu 等，全部都应小写。

【规则2-2-6】类名应该为名词或名词的组合，且各个名词的首字母大写，尽量使类名意思更明确，使用全名而尽量不要使用缩写，除非名字很长。

正例

```
class Hello;
class HelloWorld;
class Raster;
class ImageSprite;
```

【规则2-2-7】接口与类的命名约定应一致。

正例

```
interface RasterDelegate;
interface Storing;
Interface Apple;
```

【规则 2-2-8】方法名应该为动词,且第一个字母为小写,后面的每个单词首字母大写。

正例

```
run();
runFast();
```

【规则 2-2-9】如果某方法的功能是返回一个成员变量的值,方法名一般为“get + 成员变量名”,如若返回的值是布尔变量,一般以“is”作为前缀。

正例

```
getName();
isFirst();
getBackground();
```

【规则 2-2-10】如果某方法的功能是修改一个成员变量的值,方法名一般为“set + 成员变量名”。

正例

```
setBackground();
setName();
```

【规则 2-2-11】变量命名规则为首字母小写,后续单词的第一个字母大写。

正例

```
String myName;
float myWidth;
```

【规则 2-2-12】不要用“_”或“&”作为变量名的第一个字母。

【规则 2-2-13】变量名尽量使用短而且具有意义的单词。

【规则 2-2-14】单字符的变量名一般只用于生命期非常短暂的变量。

说明　i、j、k、m、n 一般用于 int 变量;c、d 一般用于 char 变量;e 一般用于例外变量。

正例

```
int i;
int n;
char c;
```

【规则 2-2-15】所有常量名均全部大写,单词间以“_”隔开。

正例

```
int MAX_NUM;
```

```
static final int MIN_WIDTH = 4;
static final int MAX_WIDTH = 999;
static final int GET_THE_CPU = 1;
```

【规则2-2-16】参数的名字必须和变量的命名规范一致。

【规则2-2-17】数组的命名方式为“byte[ ]buffer;”,而不是“byte buffer[ ];”。

【规则2-2-18】方法的参数名应使用有意义的词组,如果有可能,应使用和要赋值的字段一样的名字。

正例

```
SetCounter(int size){
    this.size = size;
}
```

第3章　注释约定

为什么要使用代码注释呢？这是因为它说明了代码的作用，即为什么要用编写该代码，而不是如何编写；同时它也指明了该代码的编写思路和逻辑方法，这样，就能使阅读者明了程序的流程及代码中的重要转折点，而不必在头脑中仿真运行代码的执行方法。

Java 程序有两种注释方法，一种是代码注释（implementation comments），另一种是文档注释（documentation comments）。代码注释类似于 C ++，用“/ * ... * /”以及“//”标识。文档注释为 Java 独有的，用“/ ** ... * /”标识，文档注释能够用 Javadoc 工具生成 HTML 形式的文档。

代码注释用于一段代码的执行说明，而文档注释用于程序的概括性描述，如一个类、方法或变量的功能描述，这些描述即使在没有代码的情况下也可以通过 Javadoc 进行浏览。注释应该能够给予代码概要性的描述，并且能够提供直接阅读代码本身所不能得到的信息，包含便于阅读和理解程序的信息。而对于包名与文件目录应该对应起来这种信息就不需要用注释来说明了。

3.1　代码注释格式

【规则 2 -3 -1】程序代码注释有 4 种方式，分别是块（block）注释、单行（single - line）注释、尾随（trailing）注释及行尾（end - of - line）注释。

【规则 2 -3 -2】块注释应该位于每个文件的开始处以及每个方法的前面。

说明　块注释用在文件、方法、数据结构以及运算法则等方面的描述。

【规则 2 -3 -3】块注释也可以用在方法的里面，但要与代码的缩排一致。块注释也应与前面代码之间有一空行。

【规则 2 -3 -4】块注释可以用“/ *”开头，也可以用“/ * -”开头。

正例

```
/ * -
   * Here is a block comment with some very special
   * formatting that I want indent(1) to ignore.
   *
   *     one
   *         two
   *             three
* /
```

【规则2-3-5】单行注释要与代码缩排一致对齐。

说明 单行注释主要用在方法内部,用于对代码、变量和流程等进行说明。

【规则2-3-6】单行注释前面一般得要有一空行。

正例

```
if(condition){
    /* Handle the condition. */
    ...
}
```

【规则2-3-7】尾随注释是很短的注释,跟代码在同一行上,但应与代码有一定的间隔。

说明 如果在一块代码中有许多这样的注释,它们必须用Tab键设置来进行缩排对齐。

正例

```
if(a == 2){
    return TRUE;                    /* special case */
} else {
    return isPrime(a);              /* works only for odd a */
}
```

【规则2-3-8】行尾注释用“//”来注释代码,可以注释一整行,也可以仅仅注释一行的一部分。

说明 它不应该用在注释连续多行的文本说明,但可以用来注释连续多行代码。

正例 下面是3个不同样式的例子。

```
if(foo > 1){

    //Do a double-flip.
    ...
}
else {
    return false;       //Explain why here.
}
//if(bar > 1){
//
//    //Do a triple-flip.
//    ...
//}
//else {
```

```
//      return false;
//}
```

【规则 2-3-9】如果已使用了规范的命名方法来创建直观、明了的代码，则可减少代码注释量。

【规则 2-3-10】如果在编码时使用了不正规的编程原则，则必须用内部注释来说明原因。

【规则 2-3-11】技巧性特别高的代码段，一定要加详细的注释。

说明　不要让其他开发人员花很长时间来研究一个高技巧但不易理解的程序段。

【规则 2-3-12】用注释来说明何时可能出错和为什么出错。

【规则 2-3-13】编写一个方法前首先写上注释。

说明　可以编写完整句子的注释或伪代码，一旦用注释对代码进行了概述，就可以很容易地在注释之间编写代码了。

【规则 2-3-14】注释应位于相关代码之前。

说明　注释一定出现在要注释的程序段前，而不要在某段程序后书写对这段程序的注释，因为先看到注释对程序的理解会有一定帮助。

【规则 2-3-15】纯色字符注释行只用于主要注释。

说明　注释中要分隔时，请使用一行空注释行来完成，不要使用纯色字符，以保持版面的整洁、清晰。

【规则 2-3-16】避免形成注释框。

说明　用星号围成的注释框，右边的星号看起来很好，但它们未给注释增加任何新的信息。实际上，这还会给编写或编辑注释的人增加许多工作。

【规则 2-3-17】注释是供人阅读的，而不是让计算机阅读的，要使用相对完整的语句。

说明　虽然不必将注释分成段落（最好也不要分成段落），但应尽量将注释写成相对完整的句子。

【规则 2-3-18】书写注释时，应避免使用缩写。

说明　缩写常使注释更难阅读，人们常用不同的方法对相同的单词进行缩写，这会造成许多混乱，如果必须对词汇进行缩写，必须做到统一。

【规则 2-3-19】对注释也应进行缩进，使之与后面的语句对齐。

说明　注释通常位于它们要说明的代码的前面。为了从视觉上突出注释与它的代码之间的关系，应将注释缩进，使之与代码处于同一个层次上。

【规则 2-3-20】为每个方法赋予一个注释标头。

说明　每个方法都应有一个注释标头。方法的注释标头可包含输入参数、返回值、原始作者、最后编辑该方法的程序员、上次修改日期及版权信息等内容。

【规则 2-3-21】在每个 if 语句的前面加上注释。

【规则 2-3-22】在每个 switch 语句的前面加上注释。

【规则 2-3-23】在每个循环的前面加上注释。

说明　用清楚、直观的注释来说明循环的作用。

【规则 2-3-24】对实参，需用注释说明参数类型、参数作用、约束或前提条件等。

【规则 2-3-25】对字段，需用注释说明字段的属性、描述信息等。

【规则 2-3-26】注释所有使用的变量。

【规则 2-3-27】注释所有的并行事件和可见性决策。

【规则 2-3-28】对类，需用注释说明创建类的目的、已知的问题、类的开发/维护历史等。

【规则 2-3-29】对每一个类/类内定义的接口，都要含有注释说明。

【规则 2-3-30】对接口，应用注释说明其目的和如何使用。

【规则 2-3-31】局部变量的注释应说明其用处和目的。

【规则 2-3-32】成员函数注释要说明成员函数的作用及其具体方法。

【规则 2-3-33】成员函数要注释各个参数及返回值。

【规则 2-3-34】任何由某个成员函数抛出的异常都必须注释。

【规则 2-3-35】必须要有修改代码的历史注释。

3.2　文档注释格式

文档注释可以被 Javadoc 处理，生成 HTML 文件。Javadoc 是 J2 里面一个非常重要的工具。如果程序员按照规范在 Java 的源代码里面写好注释，那么就可以生成相应的文档，查看起来会非常方便。很多 IDE 都可以直接生成 Java 注释文档。

【规则 2-3-36】文档注释以“/**”开头，“*/”结尾。

说明　Javadoc 通常从 package、公开类或者接口、公开或者受保护的字段、公开或者受保护的方法提取信息。

正例

```
/**
 *
 * @param id the coreID of the person
 * @param userName the name of the person
 * you should use the constructor to create a person object
 */
public SecondClass(int id,String userName)
{
    this.id = id;
    this.userName = userName;
}
```

【规则 2-3-37】文档注释不能放在方法或程序块内。

说明　文档注释应该写在要说明部分的前面，并且在其中可以包括 HTML 标记。

【规则 2-3-38】一般文档注释可以分为类注释、方法注释和字段注释。

正例

```
/**
 * The Example class provides...
 */
public class Example {...
```

注意，处在最顶层的类和方法不用缩进。

【规则 2-3-39】文档注释不应该用在方法或构造函数的内部。

说明　因为 Java 只会关联第一个声明之前的文档注释与这个声明。

【规则 2-3-40】类注释应该在 import 语句的后面、类声明的前面。

正例

```
package com. north. java;
/**
 * @author ming
 *
 * this interface is to define a method print()
 *
 * you should implements this interface is you want to print the username
 *
 * @see com. north. ming. MainClass#main(String[])
 */
public interface DoSomething
{
    /**
     * @param name which will be printed
     * @return nothing will be returned
     *
     */
    public void print(String name);
}
```

其中，“@ author”和“@ see”都是常用的注释，前者表示作者，后者表示参考的连接。

【规则 2 -3 -41】方法的注释要紧靠方法的前面，可以在其中使用“@ param”、“@ return”、“@ throws”等文档标记，且置于注释行的起始处。

说明　文档标记是一些以“@”开头的命令。

正例

```
/**
 *
 * @param i
 * @return true if ... else false
 * @throws IOException when reading the file ,if something wrong happened
 * then the method will throws a IOException
 */
public boolean doMethod(int i)throws IOException
{
  return true;
}
```

【规则 2 -3 -42】只有 public 的字段才需要注释,通常是 static 的。

说明　若定义的 protected、private 类型的变量名含义明确,则不需加注释。

正例

```
/**
* the static filed hello
*/
public static int hello = 1;
```

第4章 变量、常量

【规则2-4-1】不要直接用数值进行操作，而应该定义一个数值常量。

说明 在一些判断语句或循环语句里，可以直接用-1、0和1等数值。

【规则2-4-2】用常量来取代常数时，编译器将在编译时检查常量的有效性。如果常量不存在，编译器便将给出警告，并拒绝进行编译。

说明 这可以消除错误键入的数字带来的问题，只要常量拥有正确的值，使用该常量的所有代码就会使用该正确值。

【规则2-4-3】为常量赋予较宽的作用域。

说明 在一个应用程序中，决不应该两次创建相同的常量。如果发现需要使用一个已被定义过、但超出作用域的常量，则应拓宽原常量的作用域，而不是复制该常量。

【规则2-4-4】定义有焦点的变量。

说明 用于多个目的的变量称为无焦点(多焦点)变量。无焦点变量所代表的意义与程序的执行流程有关，当程序处于不同位置时，它所表示的意义是不同的，这样就给程序的可读性和可维护性带来了麻烦。

【规则2-4-5】只对常用变量名和长变量名进行缩写。

说明 如果需要对变量名进行缩写，一定要注意整个代码中缩写规则的一致性。例如，在代码的某些区域中使用Cnt，而在另一些区域中又使用Count，就会给代码增加不必要的复杂性。

【规则2-4-6】变量名中尽量不要出现缩写。

【规则2-4-7】使用统一的序数词。

说明 通过在结尾处放置一个序数词，就可创建更加统一的变量，使它们更易被理解，也更易被搜索。例如，应使用strCustomerFirst和strCustomerLast，而不应使用strFirstCustomer和strLastCustomer。

【规则2-4-8】使用肯定形式的布尔变量。

说明 给布尔变量命名时，始终都要使用变量的肯定形式，以减少其他开发人员在理解布尔变量所代表的意义时的难度。

【规则 2-4-9】为每个变量选择最佳的数据类型。

说明　这样做既能减少对内存的需求量、加快代码的执行速度，又会降低出错概率。变量的数据类型可能会影响该变量进行计算所产生的结果。在这种情况下，编译器不会产生运行期错误，它只是迫使该值符合数据类型的要求。这类问题极难查找。

【规则 2-4-10】尽量缩小变量的作用域。

说明　如果变量的作用域大于它应有的范围，变量可继续存在，并且在不再需要该变量后的很长时间内仍然占用资源。它们的主要问题是，其他类中的某些方法都可能对它们进行修改，并且很难跟踪究竟是在何处进行修改的。占用资源是作用域涉及的一个重要问题。对变量来说，尽量缩小作用域将会对应用程序的可靠性产生巨大的影响。

【规则 2-4-11】不要随便将某个变量或方法定义为 public 类型。

说明　如果某个方法并不希望被其他类调用，就更不能定义为 public 类型。

【规则 2-4-12】实例变量一般定义为 private 类型。

说明　此类变量可通过 get 和 set 方法进行访问，这样就能利用 Java 中 reflect 的一些特性对其进行灵活的处理。

【规则 2-4-13】public static 型的类变量和方法可直接通过类来访问。

说明　类变量和方法就是那些 public static 型的变量和方法，这些变量和方法不属于任何一个这个类的实例，所以在访问它们的时候不应该通过类的实例来访问，而应该直接通过类来访问。

正例

```
classMethod();          //正确
AClass.classMethod();   //正确
anObject.classMethod(); //错误
```

【规则 2-4-14】不要在一条语句里为不同的变量赋值。

说明　这样会给阅读带来困难。

反例

```
fooBar.fChar = barFoo.lchar = 'c';//错误
```

【规则 2-4-15】不要企图使用有嵌套的赋值语句来提高效率。

说明　应首先保证程序的清晰性。

反例

```
d = (a = b + c) + r;        //错误
```

正例

```
a = b + c;
d = a + r;
```

【规则 2-4-16】一行只应包含一个常量或变量声明。

说明 这样阅读方便,同时也便于注释。

反例

```
int level, size;
```

正例

```
int level;          //indentation level
int size;           //size of table
```

【规则 2-4-17】不要将不同类型的声明放在同一行。

反例

```
int foo, fooarray[ ];
```

【规则 2-4-18】尽量在变量声明的地方进行初始化。

说明 不过当变量需要根据后面的计算来进行赋值的时候,也可先不进行初始化。不要等到第一次使用变量时再去初始化,这样不利于理解程序,也降低了这段范围内代码的灵活性。

【规则 2-4-19】最好将声明放在每一块的开始。

说明 一块指是用{}围起来的一段代码。

正例

```
void myMethod( ) {
    int int1 = 0;           //beginning of method block

    if( condition) {
        int int2 = 0;       //beginning of if block
        ...
    }
}
```

【规则 2-4-20】在 for 循环体中可以定义变量声明。

正例

```
for( int i = 0; i < maxLoops; i ++ ) { ... }
```

【规则 2-4-21】不要在块内部定义与外部相同名字的变量。

反例

```
int count;
```

```
...
myMethod( ){
    if( condition ){
        int count = 0;      //错误
        ...
    }
    ...
}
```

【规则 2-4-22】在方法名与后面的圆括号之间以及圆括号与其后的参数列表之间都不应该有空格。

【规则 2-4-23】起始花括号“{”应该出现在声明的同一行上，且在前面有一空格。

【规则 2-4-24】结束花括号“}”应该另起一行。

说明　不过当语句为 NULL 时例外，应写成“{NULL}”的形式。

正例

```
class Sample extends Object {
    int ivar1;
    int ivar2;

    Sample( int i, int j) {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod( ) { }

    ...
}
```

第5章　表达式和基本语句

【规则 2-5-1】每行应该只有一条语句。

正例

```
argv ++ ;          //Correct
argc -- ;          //Correct
```

反例

```
argv ++ ;argc -- ;
```

【规则 2-5-2】复合语句应用花括号“{}”包起来。

【规则 2-5-3】复合语句中各语句缩排格式应统一。

【规则 2-5-4】复合语句中,“{”应该在行尾,“}”应该另起一行并且与本复合语句缩进对齐。

正例

```
if( condition) {
    statements;
}
```

【规则 2-5-5】即使复合语句中只包含一个语句,也应用“{}”。

说明　在一些控制语句中,如在 if-else 或 for 语句中应特别注意这样的问题,以便于语句的追加,也可避免因遗漏括号而引起的程序错误。

【规则 2-5-6】返回语句后一般不应该用圆括号括起来,除非不得已要让返回值看起更明显。

正例

```
return;

return myDisk. size( ) ;


return( size? size:defaultSize) ;
```

【规则 2-5-7】对 if-else 及 if-elseif 语句,在任何情况下,都应该用“{}”格式。

反例

```
if( condition )
    statement;
```

正例　if-else 语句应该有如下的格式：

```
if( condition ) {
    statements;
}

if( condition ) {
    statements;
} else {
    statements;
}

if( condition ) {
    statements;
} else if( condition ) {
    statements;
} else {
    statements;
}
```

【规则 2-5-8】for 循环语句应遵循以下格式：

```
for( initialization; condition; update ) {
    statements;
}
```

【规则 2-5-9】一个空的 for 循环语句，即其所有的工作都将在初始化、条件判断及条件更新中完成，其格式为：

```
for( initialization; condition; update );
```

说明　当在初始化、条件判断、条件更新使用逗号的时候，应当避免使用超过 3 个变量，否则句子将会变得难以理解。如果需要，可以将句子分开来写，如将条件判断放在循环之前、条件更新放在循环之后。

【规则 2-5-10】while 语句应该有以下的格式：

```
while(condition){
    statements;
}
```

【规则 2-5-11】一个空的 while 语句应该有以下的格式：

```
while(condition);
```

【规则 2-5-12】do-while 循环应该有以下的格式：

```
do {
    statements;
} while(condition);
```

【规则 2-5-13】switch 语句应该遵循一定的格式。

说明　一个 switch 语句应该有以下的格式：

```
switch(condition){
case ABC:
    statements;
    /* falls through */

case DEF:
    statements;
    break;

case XYZ:
    statements;
    break;

default:
    statements;
    break;
}
```

如果一条分支执行完并不跳出，就应该用注释说明，如“/* falls through */”。

【规则 2-5-14】每个 switch 语句都应该包含一个 default 分支。

说明　在 default 中的 break 也许是多余的，但可以避免以后添加其他分支等情况可能出现的错误。

【规则 2 - 5 - 15】try-catch 语句应该遵循一定的格式。

说明　一个 try-catch 语句应该有以下的格式：

```
try {
    statements;
} catch(ExceptionClass e){
    statements;
}
```

【规则 2 - 5 - 16】一个 try-catch 语句可能会有 finally 语句，finally 里面的语句总会被执行到。

正例

```
try {
    statements;
} catch(ExceptionClass e){
    statements;
} finally {
    statements;
}
```

第6章　类和类方法

【规则2-6-1】创建具有很强内聚力的类，即应该将相关的方法组织在一个类中。

说明　当类包含一组紧密关联的方法时，该类可以说具有强大的内聚力。当类包含许多互不相关的方法时，该类便具有较弱的内聚力。应该努力创建内聚力比较强的类。

【规则2-6-2】将一些与其他方法无关、独立性较强的方法放在一个综合性类中。

说明　大多数工程都包含许多并不十分适合与其他方法组合在一起的方法。在这种情况下，可以为这些方法创建一个综合性类。

【规则2-6-3】创建“模块化”的类。

说明　类的基本目的是创建相对独立的程序单元。

【规则2-6-4】创建高度专用的方法。所有方法都执行专门的任务。

说明　每个方法都应执行一项特定的任务，应避免创建执行许多不同任务的方法。

【规则2-6-5】创建松散连接的方法。尽量使方法成为自成一体的独立方法。

说明　当一个方法依赖于其他方法的调用时，称之为与其他方法紧密连接的方法。紧密连接的方法会使调试和修改变得比较困难，因为它牵涉更多的因素。松散连接的方法优于紧密连接的方法。

【规则2-6-6】尽量减少类变量，使方法具备较强的独立性。

说明　创建方法时，设法将每个方法视为一个黑箱，其他例程不应要求了解该方法的内部工作情况，该方法也不应要求了解它外面的工作情况。这就是为什么应使方法依靠于参数而不依靠于全局变量的原因。

【规则2-6-7】将复杂进程放入专用方法。

说明　如果应用程序使用复杂的数学公式，应为每个公式创建一个方法。这样，在使用这些公式的其他方法中，就不会包含用于该公式计算的实际代码，这种方法也可以更容易地发现与公式相关的问题。

【规则2-6-8】将方法设计成简要的、功能性单元，用它描述和实现一个不连续的类接口部分。

说明　理想情况下，方法应简明扼要。若长度很大，可考虑通过某种方式将其分割成较短的

几个方法,这样做也便于类内代码的重复使用。

【规则 2-6-9】将数据的输入/输出(I/O)处理放入专用方法中。

【规则 2-6-10】将专用方法中可能要修改的代码隔离。

说明　如果知道某个进程经常变更,应将这个多变的代码放入专用方法,以便以后可以更方便地进行修改,并减少无意中给其他进程带来问题的可能性。

【规则 2-6-11】将业务规则封装在专用方法中。

说明　业务规则常属于要修改的代码类别,应与应用程序的其余部分隔开。其他方法不应知道业务规则,只有要调用的方法才使用这些规则。

【规则 2-6-12】创建更加容易调试和维护的方法。

【规则 2-6-13】提高方法的扇入和降低方法的扇出。

【规则 2-6-14】为方法和类赋予表义性强的名字。

说明　为了使代码更加容易理解,最容易的方法之一是为方法赋予表义性强的名字。函数名 DoIt、GetIt 的可读性很难与 CalculateSalesTax、RetrieveUserID 相比。正确地命名方法,可使程序的调试和维护工作大为改观。要认真对待方法命名工作,不要为了减少键入操作量而降低方法的可理解性。

【规则 2-6-15】给方法命名时应大、小写字母混合使用,并且定义方法名时不要使用缩写。

【规则 2-6-16】应为每个方法赋予单个退出点。

【规则 2-6-17】创建方法时,始终都应显式地定义它的作用域。

【规则 2-6-18】创建一个公用方法时必须用注释加以说明。

【规则 2-6-19】如果一个方法只被同一类中的另一个方法调用,那么应将它创建成私有方法。

【规则 2-6-20】如果一个方法是被多个类中的多个方法中调用,则应将该方法定义为公用方法。

【规则 2-6-21】用参数实现方法之间的数据传递。

【规则 2-6-22】为每个参数指定数据类型。

【规则 2-6-23】始终要对数据进行检验。

说明　决不要假设使用的数据没有问题。程序员常犯的一个错误是在编写方法时假设数据没有问题。在编程初始阶段,当编写调用方法时,这样的假设并无大碍。这时完全能够知道什么是参数的许可值,并按要求提供这些值。但如果不对参数的数据进行检验,那么下列情况就有可能带来很大麻烦:由其他人创建了一个调用方法,但此人不知道允许的值;程序员添加了新的调用方法,并错误地传递了数据。

【规则 2-6-24】为了常规用途而创建一个类时,应采取一般的经典形式。

说明　它还应包含对下述元素的定义:

equals( )

hashCode( )

toString( )

clone( ) (implement Cloneable)

implement Serializable

【规则 2-6-25】对于创建的每一个类,都应考虑置入一个 main( ),其中包含了用于测试该类的代码。

说明　没必要删除类中的测试代码。若进行了任何形式的改动,都可方便地返回测试。这些代码也可作为如何使用类的一个示例使用。

【规则 2-6-26】设计一个类时,类的使用方法应该是非常明确的,以方便客户程序员的使用。

【规则 2-6-27】设计一个类时,应同时考虑将来代码维护的方便性(预计有可能进行哪些形式的修改以及代码优化等)。

【规则 2-6-28】使类尽可能短小、精悍,而且只解决一个特定的问题。

【规则 2-6-29】若一个类中包含的方法较多,且其操作类型差别也较大,则应考虑用几个类来分别实现,而不是放到一个类中。

【规则 2-6-30】如果一个类中的多个成员变量在特征上有很大的差别,则应考虑将它们分离为多个类。

第7章 代码规范

【规则2-7-1】exit()除了在main()中可以被调用外,其他的地方都不能被调用。

说明 这样做可保证任何代码都不可能截获退出。

【规则2-7-2】申明的错误应该抛出一个RuntimeException或者派生的异常。

【规则2-7-3】顶层的main()函数应该截获所有的异常,并且打印在屏幕上(或者记录在日志中)。

【规则2-7-4】Java使用成熟的后台垃圾收集技术来代替引用计数。

说明 必须在使用完对象的实例以后进行删除工作。

正例

```
FileOutputStream fos = new FileOutputStream(projectFile);
project.save(fos,"IDE Project File");
fos.close();
```

【规则2-7-5】充分利用Clone方法。

正例

```
implements Cloneable

public
  Object clone()
  {
      try {
        ThisClass obj = (ThisClass)super.clone();
        obj.field1 = (int[])field1.clone();
        obj.field2 = field2;
        return obj;
      } catch(CloneNotSupportedException e) {
        throw new InternalError("Unexpected CloneNotSUpportedException: " +
  e.getMessage());
      }
}
```

【规则2-7-6】绝对不要因为性能的原因将类定义为final类型的(除非程序框架要求)。

说明 如果一个类还没有准备好被继承,最好在类文档中注明,而不要将它定义为final类型。这是因为没有人可以保证是否需要继承它。

【规则2-7-7】Utility类(仅仅提供方法的类)应该被申明为抽象类型,以防止被继承或被初始化。

【规则2-7-8】注意初始化数组的方法。

【规则2-7-9】在Java代码中,注意合理利用枚举类型。

【规则2-7-10】在含有混合操作符的表达式中尽量使用括号。

反例

```
if (a == b && c == d);
```

正例

```
if ((a == b) && (c == d))
```

【规则2-7-11】如果一个条件表达式中在“?”之前含有二元操作符,就应该用括号括起来。

正例

```
(x >= 0) ? x : -x;
```

【规则2-7-12】充分利用Log。

说明 Log一般可分为以下几类:

- 调试用debug log, trace log
- 操作用info log
- 出错用error log

【规则2-7-13】让一切代码都尽可能地“私有”——private。

说明 如果库的某一部分(一个方法、类或者一个字段等)“公共化”了,就不能轻易将它们删除。若强行删除,就可能破坏其他人现有的代码,使他们不得不重新编写和设计。若只公布自己必须公布的,将能有效地降低在改变其他代码时发生错误的几率。在多线程环境中,隐私是特别重要的一个因素——只有private字段才能在非同步使用的情况下受到保护。

【规则2-7-14】绝对杜绝“巨大对象综合症”。

说明 对一些习惯于顺序编程思维、且初涉OOP领域的新手,往往喜欢先写一个顺序执行的程序,再把它嵌入一个或两个巨大的对象里。

【规则 2-7-15】任何时候只要发现类与类之间结合得非常紧密,就要考虑是否采用内部类,从而改善编码及维护工作。

【规则 2-7-16】常数在代码中必须以常量形式出现。

说明　避免使用“魔术数字”,这些数字很难明确的表达其含义,如根本不知道“100”到底是指数组大小还是代表其他含义。所以,应创建一个常量,并为其使用具有说服力的描述性名称,并在整个程序中都采用常量标识符,这样可使程序更易理解及维护。

【规则 2-7-17】在代码中,应按功能逻辑用空行将代码分段。

【规则 2-7-18】在代码中应尽量书写短的、功能单一的语句。

【规则 2-7-19】合理组织代码,将同一类代码放在相对集中的位置,提高敏感度。

反例

```
anObect. message1( );
anObect. message2( );
aCount = 1;
anObect. message3( );
```

正例

```
anObect. message1( );
anObect. message2( );
anObect. message3( );

aCount = 1;
```

第8章　其他规范

【规则2-8-1】代码中要考虑错误处理和异常事件处理。

说明　程序运行过程中出现的错误和异常事件包括逻辑和编程错误、设置错误、被破坏的数据、资源耗尽等。

系统在正常状态下以及无重载和硬件失效状态下，不应产生任何异常。异常处理时可以采用适当的日志机制来报告异常，包括异常发生的时刻。

【规则2-8-2】不要使用异常来控制程序流程。

【规则2-8-3】当抛弃捕获的异常处理时，应检查对象的创建。

说明　涉及构建器和异常的时候，通常希望丢弃在构建器中捕获的任何异常，包括对象创建失败，从而保证调用者进行正确的操作。

【规则2-8-4】非商务公用组件应单独封装。

【规则2-8-5】每一个业务流程都应单独封装。

【规则2-8-6】一次方法(组件)的调用应能完成某一项功能或流程，即符合完整性。

【规则2-8-7】一次方法(组件)的调用应符合ACID特性。

说明　ACID即Atomicity(原子性)、Consistency(一致性)、Isolation(隔离性)和Durability(持久性)，它与事务的概念密切相关，可支持多用户同时访问一个文件。

【规则2-8-8】多次方法(组件)的调用应包含在一个事务中。

【规则2-8-9】尽量不要使用已经被标为不赞成使用的类或方法。

【规则2-8-10】如果需要换行的话，尽量用“println”来代替在字符串中使用“\n”。

反例

```
System.out.print("Hello,world! \n");
```

正例1

```
System.out.println("Hello,world!");
```

正例2　也可以构造一个带换行符的字符串，来实现输出换行。

```
String newline = System.getProperty("line.separator");
System.out.print("Hello world!" + newline);
```

【规则 2-8-11】用 separator()方法代替路径中的“:”或“;”。

【规则 2-8-12】用 pathSeparator()方法代替路径中的“/”或“\”。

【规则 2-8-13】尽量不要将 AWT 组件和 Swing 组件混合起来使用。

【规则 2-8-14】AWT 组件绝对不能用 JscrollPane 类来实现滚动。滚动 AWT 组件的时候一定要用 AWT ScrollPane 组件来实现。

【规则 2-8-15】避免在 InternalFrame 组件中使用 AWT 组件。

【规则 2-8-16】AWT 组件总是显示在 Swing 组件之上。

说明　当使用包含 AWT 组件的弹出式菜单时要小心,尽量不要这样使用。

【规则 2-8-17】在软件调试中使用配置开关。

【规则 2-8-18】一种调试方法就是用一个 PrintStream 类成员。

说明　在没有定义调试流的时候就为 NULL,类要定义一个 DEBUG 方法来设置调试用的流。

【规则 2-8-19】在写代码的时候,应始终考虑性能问题。

说明　这不是说时间都应该浪费在优化代码上,而是应该时刻提醒自己要注意代码的效率。例如,在编程时没有时间来实现一个高效的算法,那么就应该在文档中记录下来,以便在以后有空的时候再来实现。

【规则 2-8-20】不要在循环中构造和释放对象。

【规则 2-8-21】在处理 String 时要尽量使用 StringBuffer 类,StringBuffer 类是构成 String 类的基础。

说明　String 类将 StringBuffer 类封装了起来(以花费更多时间为代价),它为开发人员提供了一个安全的接口。在构造字符串时,应该用 StringBuffer 来实现大部分的工作,当工作完成后将 StringBuffer 对象再转换为需要的 String 对象。例如,如果有一个字符串必须不断地在其后添加许多字符来完成构造,那么就应该使用 StringBuffer 对象和 append() 方法。如果用 String 对象代替 StringBuffer 对象,则需要花费许多不必要的创建和释放对象的 CPU 时间。

【规则 2-8-22】避免太多地使用 synchronized 关键字。

说明　只在必须时使用关键字 synchronized,这是一个避免死锁的好方法。

【规则 2-8-23】PrintStream 已被约定为不赞成使用,应用 PrintWriter 来代替它。

【规则 2-8-24】当客户程序员用完对象以后,若要对类进行清除工作,可考虑将清除代码置于一个良好定义的方法里,可采用类似于 cleanup()这样的名字,明确表明其用途。

说明　除此以外,还可在类内放置一个布尔类型变量作为标记,指出对象是否已被清除。在类的 finalize()方法里,应确定对象已被清除,并已丢弃了从 RuntimeException 继承的一个类(如果还没有的话),从而指出一个编程错误。在采取此方法之前,应确认是否能在自己的系统中运行 finalize(),因为它可能需要调用 System. runFinalizersOnExit(TRUE)函数。

【规则 2-8-25】在一个特定的作用域内,若一个对象必须清除(不是由垃圾收集机制处理),可采用下述方法:初始化对象;若成功,则立即进入一个含有 finally 从句的 try 块,开始清除工作。

【规则 2-8-26】若在初始化过程中需要覆盖(取消)finalize(),一般应调用 super. finalize()(若 Object 属于直接超类,则无此必要)。

说明　在对 finalize()进行覆盖的过程中,对 super. finalize()的调用应属于最后一个行动,而不应是第一个行动,这样可确保在需要基础类组件的时候它们依然有效。

【规则 2-8-27】创建大小固定的对象集合时,可以采用创建数组的方法(若准备从一个方法里返回这个集合,更应如此操作)。

说明　这样做就可享受到数组在编译期进行类型检查的好处。

【规则 2-8-28】尽量使用 interfaces,不要使用 abstract 类。

说明　在准备创建一个基础类时,首先应考虑能否定义为一个 interface(接口)。只有在不得不使用方法定义或者成员变量的时候,才需要将其定义为一个 abstract(抽象)类。接口主要描述了客户希望做什么事情,而一个类则致力于(或允许)具体的实施细节。

【规则 2-8-29】对象不应只是简单地容纳一些数据;它们的行为也应得到良好的定义。

【规则 2-8-30】在已定义类的基础上创建新类时,尽可能避免使用继承。

说明　除非设计上有要求,否则应避免使用继承,因为它会增加程序的复杂性。

【规则 2-8-31】为避免编程时遇到麻烦,应保证在自己类路径指到的任何地方,每个名字都仅对应一个类。

说明　否则，编译器可能先找到同名的另一个类，并报告出错消息。若怀疑自己碰到了类路径问题，可通过在类路径的每一个起点搜索同名的.class 文件来检查。

【规则 2-8-32】用合理的设计方案消除“伪功能”。

【规则 2-8-33】警惕“分析瘫痪”。

说明　请记住，无论如何都要提前了解整个项目的状况，再去考察其中的细节。由于把握了全局，可快速认识自己未知的一些因素。

【规则 2-8-34】警惕“过早优化”。

说明　编制程序时，首先考虑的是如何正确实现其功能，第二步才能考虑其性能上的优化。应借助专门的分析工具寻找代码中存在的性能瓶颈，对其进行优化。而且优化工作应特别谨慎，因为有时性能提升的隐含代价是使代码变得难于理解及维护。

【规则 2-8-35】阅读代码的时间一般比写代码的时间多得多。

说明　思路清晰的设计可获得易于理解的程序，但注释及一些示例往往具有不可估量的价值。无论对自己，还是对以后阅读本程序的人，它们都是相当重要的。

【规则 2-8-36】每完成一个模块，就应进行测试或代码审查，这样能及时发现问题。

说明　采取这种方式，往往能在最适合修改的阶段找出一些关键性的问题，避免产品发行后再解决问题而造成的金钱及精力方面的损失。

【规则 2-8-37】应尽量参考、利用“成熟”的代码。

第三部分

Delphi 语言编程规范

第1章　程序约定

1.1　项目总体及布局规则

【规则3-1-1】每一个项目的代码、文档按模块、功能必须在项目文件夹中有条理地归类存放。

说明　每个项目文件夹(建议)包含以下子文件夹:

① \Bin:它是产品的工作目录,用于存放项目中所有可执行文件的当前版本。

② \ SQL:用于存放数据库的 SQL 文件(只针对数据库程序的开发)。

③ \Lib:用于存放与项目有关的库文件。

④ \Man:它包括了项目的所有外部文档。包括手册、帮助文件、其他在线文档、README 文件以及其他将和产品一起发放到用户手中的文档。

⑤ \Res:包括应用程序的所有共享资源,如 Icon(图标)、资源文件、Bitmap 等。

⑥ \Include:包含公用的窗体或单元。

⑦ \Control:用于存放项目内用的自编或第三方提供的控件。

⑧ \Source:用于存放程序源代码。

⑨ \Public:存放可以公用的模块或程序。

⑩ \ Doc:存放文档文件。

⑪ \Hlp:存放帮助文件。

⑫ \Backup:用于存放备份文件。

⑬ \Tmp:用于存放临时文件。

【规则3-1-2】每个项目的主目录下均必须有一个项目说明文件,说明该项目的一些概要性提示和相关规范。

【规则3-1-3】在项目文件夹下的每个子目录中必须有一个文件夹说明文件,说明该层文件夹及其子文件夹的分类方法和含义。

说明　任何时候,新增一个文件夹时,均必须在同层目录下的文件夹说明文件中添加所新增文件夹的分类含义,同时创建该文件夹下的文件夹说明文件。

【规则3-1-4】任何一个工程文件(包括动态链接库工程文件)的第一部分必须用注释的形式说明项目名称、公司版权、工程描述、版本说明、创建日期、作者以及后续更新人员。

【规则3-1-5】在Delphi中,按照项目为单位来组织程序文件。

说明　一个典型的项目应包括:

① 项目文件(.DRP文件):含有工程主程序的Pascal源代码。

② 单元文件(.PAS文件):项目中每个窗体的Pascal源文件,包含该窗体的所有声明和过程(包括时间处理过程)。

窗体文件(.DFM文件):含有一个窗体设计属性的二进制文件,每个窗体的.DFM文件与.PAS文件相互对应。

资源文件(.RES文件):编译的二进制资源文件,被链接到应用程序的可执行文件中。

项目选项文件(.DOF文件):存储了"Project→Options"菜单命令所设置的项目选项。

桌面设置文件(.DSK文件):存储了"Tools→Options"菜单命令所设置的桌面选项。

包文件(.DPK/.BPL文件):用于共享组件、类、数据和代码的文件。源文件为.DPK文件,编译后为.BPL文件。

【规则3-1-6】在所有源文件、工程文件、单元等中使用信息化文件头,其中首先必须用注释的形式说明项目名称、公司版权。

【规则3-1-7】在文件头后以注释的形式说明模块名称、模块描述、模块版本、创建日期、作者、更新人以及TODO列表。

正例

```
{…
修改日期:
作者:
用途:
本模块结构组成:
}
```

【规则3-1-8】在Interface部分USE单元原则上只允许Delphi的IDE自动添加,如确需自己添加,则必须在引用到的单元名后用"{}"注释添加原因。用于编译开关控制的伪指令应插入在USE之前。

【规则3-1-9】Interface部分应当只包含需要被外部单元访问的类型、变量、过程和函数的声明,而且这些声明应当在Implementation部分之前。

【规则3-1-10】Implementation部分包含本单元私有的类型、变量、过程和函数的声明。

【规则3-1-11】不要在Initialization部分写过多代码。如有代码,应详细说明必须写在Initialization部分的原因。

【规则 3 -1 -12】在 Finalization 部分释放所有在 Initialization 部分中分配的资源。

【规则 3 -1 -13】除主 Form 外，每个 Form 单元都应含有实例化函数(入口函数)，用于创建、设置、显示和释放 Form。

说明　Form 的变量不应放在单元中，而应作为实例化函数中的局部变量(应首先从“Project Options”对话框的自动生成列表中移走该 Form)。

【规则 3 -1 -14】采用逐级缩进形式来书写源程序。缩进的规则为每一级缩进两个空格。

说明　每级之间要缩进两个空格，这样会使程序层次分明、错落有致。不要将“TAB”符号存储到源文件中，原因是“TAB”字符在不同的源代码管理器中宽度会有不同的定义。在“Tools→Editor Options”的“General”页中不选择“Set Tab Character”和“Optimal Fill”复选框，可使制表符“TAB”不被保存。

正例　当遇到 begin 或进入判断、循环、异常处理、with 语句、记录类型声明、类声明等的时候增加一级，当遇到 end 或退出判断、循环、异常处理、with 语句、记录类型声明、类声明等的时候减少一级。

【规则 3 -1 -15】行宽应保持默认设置的 80 个字符，只要可能，长度超过 80 个字符的语句必须在逗号或运算符处换行。换行后，应缩进两个字符。

说明　Delphi 编辑器在右边大约第 81 个字符处留有一条暗线，实际上在 Delphi 的默认界面下，当分辨率在 800 ×600 时，最大化窗口将显示到该暗线左边 4 个字母处。因此，不要将源代码写到暗线之外，也就是说每行包括空格不要多于 80 个字符。如果语句过长，那么应换行写，换行后要缩进两个字符。这样也易于打印，在 Delphi 中超过暗线的部分不会被打印出来。如果使用 Word 等文字处理软件打印 Delphi 程序，超出的部分将会被调整到下一行的首部，这样打印出的程序将难以阅读。所以，尽量在编写代码的时候做好一切调整，不要把这种工作留到打印的时候再进行。

【规则 3 -1 -16】换行时要注意保持程序的可读性，尽量保持完整的部分。如果函数过长，那么在换行时应将某个完整的参数说明换行，而不要只将数据类型声明换行书写。

反例

```
function AdditonFiveInputNumber(a: integer; b: integer; c: integer; d:
  ineger; e: integer): integer;                    //错误
```

正例

```
function AdditonFiveInputNumber(a: integer; b: integer; c: integer; d: ineger;
  e: integer): integer;                            //正确

function AdditonFiveInputNumber(
  a: integer;
```

```
b: integer;
c: integer;
d: ineger;
e: integer): integer;                    //正确
```

【规则3-1-17】应在类、方法、过程及函数之间以及算法段间使用空行分隔。空行只能用一行,多余的空行必须删除。

【规则3-1-18】空格的使用是为了保持程序的整洁,使程序员能够快速了解程序结构,其使用也需遵循一定的规范。

说明　下面是一些规范和相应的例子:

① 每个单词之间要留有一个空格。例如:

```
for TMyClass = class(TObject)
```

② 在“=”、“< >”、“>=”、“<=”左右要留有一个空格;在“:=”和“:”右边要留有一个空格,而左边不留。例如:

```
if a = b then a:= b;a: integer;
```

③ 保留字和关键字与左边的符号间要留有一个空格,与右边的符号间不留。例如:

```
procedure ShowMessage; overload;
```

④ 括号的使用:在过程和函数的定义和调用中,括号与外部的单词和符号之间不留空格;与内部的单词之间不留空格。在if语句的条件判断中,与and、or等保留字之间要使用空格。例如:

```
function Exchange(a: integer; b: integer);
if (a = b) and ((a = c) or (a = d)) then … end;
```

⑤ 在操作符及逻辑判断符号的两端添加空格,例如:

```
I := I + 1;
a and b
```

但添加括号时不需要空格。例如:

```
if ( a > b ) then   //错误的用法
If (a > b) then   //正确的用法
```

【规则3-1-19】begin和end语句要独占一行,begin要与上一层的第一个字母对齐,也就是说在换行之后不要留任何空格而直接写begin,end要与所对应的begin对齐。

反例

```
for I := 0 to 10 do begin //不正确的用法
end;
```

正例　下面是正确的写法:

```
for i := 1 to 10 do
begin
```

```
    ...
    end;
```

但在 if 语句中有个特例,如下:

```
    if Condition then
    begin
    ...
    end
    else begin
    ...
    end;
```

这样写的目的是要保证程序足够紧凑。语句分层的好处是使程序段清晰,但是过分地、不尽合理地分层会使程序过于松散,这同样是需要避免的。

【规则 3-1-20】除主模块、公共函数模块和公共数据模块外,所有该项目下的单元不可由项目自动创建(Create)。在加入新单元后,必须在工程文件中删除这些自动创建的语句。

1.2 命名约定

【规则 3-1-21】所有的自定义名称中的每个单词的首字母要使用大写,其他字母使用小写。Delphi 保留字和关键字要全部使用小写。Delphi 预定义函数名的写法与自定义名称写法相同。Delphi 中的基本数据类型名要使用小写,扩展的类类型名的前两个字母要大写(类类型的首字母是"T")。

正例

```
MyFavouriteSong, CarList;//自定义名称
if (a = b) and ((a = c) or (a = d)) then ··· end;//保留字
ShowMessage('Everything all right');//Delphi 预定义函数名
MyStrings = TStrings.Create;//Delphi 扩展类类型名
```

【规则 3-1-22】命名的基本原则是名称要能够明确表示数据的功能,应使用动词、名词或二者的组合。Object Pascal 支持长文件名。

【规则 3-1-23】绝对不可以使用 Delphi 中定义的保留字和关键字,而且尽量不要使用其他语言中定义的保留字和关键字。

【规则 3-1-24】尽量使用完整的词语命名,避免或减少使用缩写、下画线或其他符号。

【规则 3 -1 -25】要注意对布尔变量的命名。布尔变量的名称要能够明确地表示出 TRUE 和 FALSE 的含义。

说明　如为记录某文件是否存在的变量命名时,使用 IsFileExisted 就比使用 FileExisted 好。

【规则 3 -1 -26】永远不要将一个全局变量命名为 Temp 或 Tmp。

说明　如果作为全局变量,很可能会出现类型不匹配的赋值语句,虽然此时编译器会给你很大的帮助,但是难以避免细小错误的发生。但是在过程或函数中如此命名有时还是允许的。因为很多时候这样命名的确很方便。

【规则 3 -1 -27】过程和函数的名称应全部使用有意义的单词组成,以大写的字母开头,且中间部分的单词也应适应使用首字母大写形式,以增强其可读性。

反例

```
procedure thisismuchmorereadableroutinename;
```

正例

```
procedure FormatHardDisk;          //正确的命名
```

【规则 3 -1 -28】例程名应当有意义,并应尽量使用动宾词组。

说明　例程的名称应该同它的内容相符。一个会导致某个行为的例程应以动词开头。

正例

```
procedure FormatHardDrive;
```

【规则 3 -1 -29】一个用于设置输入参数的例程应以单词“set”作为前缀。

正例

```
procedure SetUserName;
```

【规则 3 -1 -30】一个用来接收某个值的例程应以单词“get”作为前缀。

正例

```
function GetUserName: string;
```

【规则 3 -1 -31】为过程和函数的参数命名时,同一类型的参数应写在同一行中。

正例

```
procedure Foo(Param1,Param2,Param3: Integer; Param4: string);
```

【规则 3 -1 -32】为过程和函数的参数命名时,所有参数必须是有意义的;并且当参数名称和其他属性名称重复时,应加一个前缀“A”。

正例

```
procedure SomeProc(AUserName: string; AUserAge: integer);
```

【规则 3 -1 -33】命名冲突。若使用的两个 unit 中包括一个重名的函数或过程，在引用这一函数或过程时，将执行在 use 子句中后声明的那个 unit 中的函数或过程。这就称为产生了命名冲突。为了避免这种情况的发生，应在引用函数或过程时，将函数或过程的出处写完整。

正例

```
SysUtils. FindClose(SR);
Windows. FindClose(Handle);
```

【规则 3 -1 -34】Delphi 的 IDE 自动产生的事件句柄、名称和参数不可作改动。

【规则 3 -1 -35】变量名(包括程序中文件名、常量名、过程名和函数名等)的命名规则，建议以匈牙利规则为参考，采用“限制 + 类型 + 名称”的命名方式。

说明　① 限制是指变量与常量的区分和对作用域的确定，默认为变量和局部对象。对于常量要特别指出，对于非局部变量，要表明其使用范围，如全局或模块级。

② 类型以缩写表明该程序对象的类型，如 32 位有符号整数类型和列表框控件类型。

③ 名称为对象的具体含义，要准确表达其用途，而不要使用与变量所代表的实体没有任何联系的名字，以英文、英文缩写组合给出；名称的书写采用大、小写结合的方式，如 CaseCount 表示事件计数、DeleteUser 表示删除人员等。

【规则 3 -1 -36】变量名不宜过短，也不宜过长，除去限制和类型外，以 8 ~ 15 个字符为宜。

说明　过短的命名往往不能准确描述用意，如 nCount 命名含义太不明确，可以是雇员的计数，也可以是找到的文件的计数，而命名为 nEmployeeCount 和 nFileCount 将更有助于记忆和理解；过长的命名如 recLoginUserInformation 又显得冗长，改为 recLoginUserInfor 或 recLoginUserInfo 则会在不损失含义的情况下减少输入量。

【规则 3 -1 -37】所有变量名必须是有意义的字符组合，应使其他人可以很容易读懂变量所代表的意义，变量命名可以采用同义的英文命名，可使用几个英文单词组合而成，但每一单词的首字母必须大写。

正例

```
var
WriteFormat: string;
```

【规则 3 -1 -38】变量的名称应当能表达出它的用途。尽量不使用缩写形式，并使用名词作为变量名。

【规则 3 -1 -39】变量名应该用类型的缩写作为前缀，即“限制”如 strMasterSql。

【规则3-1-40】循环控制变量可以是单个字母，如i、j、k，也可以使用具有含义的名称，如UserIndex。

【规则3-1-41】多层循环控制变量不应使用单个字母，如i、j、k等，各层的循环变量应尽可能使用有意义的名称，以避免使用时由于输入错误而产生的逻辑错误。

说明 尤其是在访问多维数组变量时，遵照此条规则可以大大减少因数组下标指定失误而引发的程序逻辑失败的几率，如将Cells[i,j]误作Cells[j,i]，而Cells[nRow,nCol]就不易产生Cells[nCol,nRow]的错误。

【规则3-1-42】在过程中使用的局部变量应遵循所有的变量命名规则。

【规则3-1-43】尽量在工程中不使用全局变量，如必须使用全局变量则必须加前缀“g”。

正例

```
gUserCount: point;    //名称为 UserCount 的全局变量
```

【规则3-1-44】在单元模块内部可以使用全局变量。所有模块内全局变量必须以“F”为前缀。如果几个模块之间需要进行数据交换，则需要通过声明属性的方法来实现。

【规则3-1-45】定义枚举类型名时应加前缀“T”，枚举类型的标识符列表的前缀应包含2~3个小写字符，它们是该枚举类型名所包含的各单词首字母组合。

正例

```
TSongType = (stRock, stClassical, stCountry, stAlternative, stHeavyMetal);
TAlign = (alNone, alTop, alBottom, alLeft, alRight, alClient, alCustom);
```

【规则3-1-46】数组类型名应表达出该数组的用途，类型名必须加字母“T”为前缀。如果要声明一个指向数组类型的指针，则必须加字母“P”为前缀，且应声明在类型声明之前。记录类型名的命名方法同上。

正例1

```
type
  PCycleArray = ^TCycleArray;
  TCycleArray = array[1..100] of integer;
```

数组类型的变量命名方法为将其类型名去掉前缀“T”。

正例2

```
type
  PEmployee = ^TEmployee;
  TEmployee = record
    EmployeeName: string
```

```
        EmployeeRate: Double;
    end;
```

【规则 3-1-47】类型名应能够体现这个类的用途，它必须以字符“T”为前缀。类型的实例名字通常和类型名相匹配而没有字符“T”。

说明　在 Delphi 中，所有的类的祖先类都是 TObject，在定义类中要明确表示出类的祖先类。也就是说，即使该类的祖先类是 TObject，也要写明（在 Delphi 中如果不写明类的祖先类，那么就默认其祖先类为 TObject）。常用的类命名格式如表 3.1.1 所示。

表 3.1.1　常用类的参考命名格式

类名	命名格式
TObject	objXXXX
TPersistent	psstXXXX
TComponent	cmptXXXX
TControl	ctrlXXXX
TWinControl	wctlXXXX
TForm	frmXXXX
TList	listXXXX
TStringList	slstXXXX
TmenuItem	miXXXX
TlistItem	liXXXX
TtreeNodes	tndsXXXX
TtreeNode	tndXXXX

正例

```
type
  TCar = class(TObject)
  private
    ...
  protected
    ...
  public
    ...
  end
```

【规则 3-1-48】当类的实例只有一个实例时，就使用没有前缀“T”的类名；如果有多个实例，那么就应另外添加其他合适的单词。

正例

```
var
Customer:TCustomer;
...
var
   FistCustomer:TCustomer;
Second Customer:TCustomer;
```

【规则 3 -1 -49】类的私有数据域(Fields),即字段命名规则和其他的变量一样,但必须在前面加上字符“F”。

说明　另外,字段名称应该为名词,而且要注意单词复数的使用(数组字段应当是复数,表示集合含义的字段也应该使用复数名称)。

所有的字段必须为私有,这样就可以通过属性来决定该字段在类的作用域之外的访问属性。这样做的目的是保证类的封装性。

正例　字段名称的排列要参照字段的含义。参考名片管理系统,对一张名片人们首先关心的是姓名、性别、年龄,然后是各种联系方式,如果更细致一点就要了解生日、家庭成员等信息。下面就是一个简单的类的定义:

```
type
TBusinessCard = class(TObject)
private
   FName: string;
   FSex: string;
   FAge: integer;
   FEmail: string;
   ...
public
   ...
end;
```

【规则 3 -1 -50】类的方法命名和过程、函数的命名一样。

说明　对于读写某个字段的方法要使用“Get”或“Set”前缀加去掉前缀“F”的字段名作为名称,前缀“Get”表示读,“Set”表示写。如果希望某个字段具有只读属性,那么需将其定义为以“Get”为前缀的方法并在属性中将其关联。

【规则 3 -1 -51】方法的参数尽量以字母 A 为前缀。

说明　如果某个方法使用名为 AName 的参数,那么此方法很可能要用到字段 FName 或属性 Name,这样在实现代码中就不会将它们混淆。

【规则 3 -1 -52】在方法的定义中要将同样用途或实现同一目标的方法分为一组，在一组的第一个方法的前面和最后一个方法后面各留一个空行。

【规则 3 -1 -53】类的属性名应该是名词，而不是动词。属性的命名规则与类的私有数据域相同，但不要加“F”前缀。

说明　属性表示的是数据，而方法表示的是行为。一般情况下属性的名称应为单数，数组类型的名称应为复数。

【规则 3 -1 -54】类的事件命名应以 ON、BEFORE、AFTER 作为前缀。

【规则 3 -1 -55】工程文件应该赋予有意义的名字。

说明　可以取组成工程名全称的各单词首字母或其缩写，也可以直接用单个单词命名。

正例　Delphi 中 Bug 管理程序的工程名字是 ddgbugs. dpr。Delphi 中系统信息的程序被命名为 SysInfo. dpr。

【规则 3 -1 -56】窗体文件的名称必须能表达 Form 的用途，并且具有 Form 前缀。

【规则 3 -1 -57】Data Module 文件的名称必须能表达数据模块的用途，并且具有“DM”前缀。

正例　用户 Data Module 文件被命名为 DMCustomers. dfm。

【规则 3 -1 -58】Remote Data Module 文件的名称必须能表达远程数据模块的用途，并且使用“RDM”作为前缀

正例　用户 Remote Data Module 文件被命名为 RDMCustomers. dfm。

【规则 3 -1 -59】通用单元（如公共函数库、变量库、常量库等）的命名应尽量表达它的含义，并以“LIB”作为前缀。

【规则 3 -1 -60】Form 实例的名称与相应的类型名称相同，但没有前缀“T”。

正例

Form 类型名：TFormAbout，　Form 实例名：FormAbout；
Form 类型名：TFormMain，　Form 实例名：FormMain；
Form 类型名：TFormCustomerEntry，　Form 实例名：FormCustomerEntry。

【规则 3 -1 -61】对于自动创建的窗体，除非特别原因，只允许主窗体自动生成。应通过“Project Options”对话框中的自动生成列表将其他自动生成的窗体删除。

【规则 3 -1 -62】数据模块的类型名称应为：前缀“TDM” + 描述性名称。

正例

```
TDMCustomer = class(TDataModule)
TDMOrders = class(TDataModule)
```

【规则3-1-63】数据模块实例的名称与相应的类型名称相似,但没有前缀“T”。

正例

```
数据模块类型名:TDMCustomerDataModule
数据模块实例名:DMCustomerDataModule
数据模块类型名:TDMordersDataModule
数据模块实例名:DMordersDataModule
```

【规则3-1-64】控件命名标准和类的命名标准相似。不同的是它们有3个字符的标志性前缀,用来表明公司、个人或其他实体。作为前缀的3个字符要用小写。

正例 一个时钟控件的名称可定义为:

```
tddgclock = class(TComponent)
```

另一个控件可以这样定义声明:

```
tlxSchool = class (TComponent)
```

【规则3-1-65】控件单元只能含有一个主要控件

说明 这是指出现在控件选项板上的控件。其他辅助性的控件或对象也可以包含在同一单元中。

【规则3-1-66】控件的注册单元名称应为前缀“Reg”。加3个字符“3个字符”即控件名缩写。

说明 控件的注册过程应从控件本身的单元中产生,并放入到一个独立的单元中。这个注册单元可以用来注册任何控件、属性编辑器、控件编辑器、专家器等。

控件的注册只应在设计时刻包中进行,注册单元应包含在设计时刻包中而不应放在运行时刻包中。

正例

若控件名称为Image MainMenu,则推荐使用的注册单元的名称为RegImm.pas。

【规则3-1-67】控件实例命名规则:控件类型前缀 + 描述性名称。

说明 所有的控件都应取个描述性的名称。使用前缀而不使用后缀的原因是在搜寻时,在对象检查器和代码探索器中搜寻控件的名字比搜寻控件的类型更容易实现。

控件类型前缀多是表现控件类型的字母缩写。如btn是TButton的缩写。前缀缩写的一般规则是,首先删除控件类型名(如TButton)的前缀“T”(得到Button),然后将除首字母外的所有元音字母均删除(得bttn),再将结果中连在一起的相同字母合并为一个字母(得btn)。如果控制类型名为由两个及两个以上单词组成的组件名,则其前缀缩写方法为删除前缀“T”后取各单

词首字母,如将 TListBox 缩写为 lb。描述性名称用于对控件意图的描述。例如,一个用于新建窗体的 TButton 控件可命名为 btnNewForm,一个编辑学生的控件可命名为 edtStudent。

【规则 3-1-68】在命名中恰当使用反义词,可以提高可读性。

说明　下面是一些常用的容易理解的词对:

Add/Remove	Begin/End	Create/Destroy
Insert/Delete	First/Last	Get/Release
Get/Set	Get/Put	Increment/Decrement
Lock/Unlock	Min/Max	Next/Previous
Next/Prior	Old/New	Open/Close
Show/Hide	Source/Destination	Source/Target
Start/Stop	Write/Read	

【规则 3-1-69】对于简单类型变量,可按一定格式缩写命名。

说明　详见表 3.1.1。

表 3.1.1　简单类型变量名缩写格式

类型	缩写	含义
Boolean	b	Boolean 变量
ByteBool	bb	Boolean Byte 变量
WordBool	bw	Boolean Word 变量
LongBool	bl	Boolean Long 变量
Integer	n	Number 变量
ShortInt	nt	Number Tiny 变量
SmallInt	ns	Number Short 变量
LongInt	nl	Number Long 变量
Int64	ne	Number Extended 变量
Comp	ne	Number Extended 变量
Byte	unt	Unsigned Number Tiny 变量
Word	uns	Unsigned Number Short 变量
LongWord	unl	Unsigned Number Long 变量
Real	f	Float 变量

续表

类型	缩写	含义
Single	fs	Float Single 变量
Double	fd	Float Double 变量
Extended	fe	Float Extended 变量
Char	c	Char 变量
AnsiChar	ca	Char Ansi 变量
WideChar	cw	Char Wide 变量
String	s	String 变量
ShortString	ss	String Short 变量
AnsiString	sa	String Ansi 变量
WideString	sw	String Wide 变量
Variant	v	Variant 变量
Enumerate	e	Enumerate 变量
Pointer	p	Pointer 变量

【规则 3-1-70】结构型变量，也按一定格式缩写命名。

说明　详见表 3.1.2。

表 3.1.2　结构型变量缩写格式

类型	缩写	含义
Array	a	Array 变量
Set	m	Mass 变量
Record	r	Record 变量
Class	o	Object 变量

【规则 3-1-71】Delphi 标准控件的命名见附件 2。

【规则 3-1-72】记住一些常见布尔型变量的命名，对规范化的实施也会有不少帮助，如 Done、Error、Found、Success、Ready 等。

1.3　注 释 约 定

【规则 3 -1 -73】Delphi 支持两种注释:块注释({})和单行注释(//)。

说明　注释的作用是为了解释程序的设计思路,说明变量及函数等代码的作用。这实际上是为了解决记忆问题,在程序设计中永远不要过分依赖大脑,而要尽可能借助文字。所以说,注释在程序设计语言中是十分重要的方面。

注释的另外一个应用是在程序调试阶段,如有两个语句,事先并不知道哪一个更好,于是需要测试:将一条语句前放置“//”(也就是说将这条语句改为注释),运行另一条语句,然后再做相反的工作,就可以轻松做出选择。

【规则 3 -1 -74】如果是一组语句,必须用块注释形式。一定注意要将“{”和“}”放在较明显的位置上,比如说放在单独的上、下两行。

【规则 3 -1 -75】多数情况下,在自定义变量、类型的前面放置注释是有必要的。

【规则 3 -1 -76】在单元文件顶部放置注释是必要的,包括“模块说明”及“单元说明”注释。

说明　注释中应包含文件名、创建日期、修改日期、作者、修改作者以及必要的描述。

正例

```
{**************************************************
项目:
文件:
功能描述:
版本:
日期:
作者:
更新:
更新日期:
更新描述:
**************************************************}
```

【规则 3 -1 -77】注释要有意义,不要使用没用的注释。

反例

```
while i < 8 do
begin
  ...
  i := i + 1;    //Add one to i
```

```
    end;
```

注释“//Add one to i”在此是毫无意义的，当然并不是说简单的语句（类似于“i := i + 1”）就不需要注释。因为简单的语句往往会起到十分重要的作用，所以，如果这条语句会使人产生疑问或者让人难以理解，那么就要将此语句的作用用注释形式写清楚。

【规则 3-1-78】不要试图在注释中创建组合图案，除非你认为这十分必要。

说明　因为在保持图案完整、美观的前提下修改注释是非常困难的。

【规则 3-1-79】要区分临时注释和永久注释。在项目完成或完成某一功能模块后，应删除临时注释。

说明　程序员用自己的方法在注释中放置特殊符号来区分代码，这样的好处是易于查找，但在编码完成后应及时将其删除。

【规则 3-1-80】对语句的更改也应反映到相应的注释中。

【规则 3-1-81】注释与代码间要留有明显的间隔，要一眼就能够分清楚哪是语句、哪是注释。

说明　可以将注释放在代码行的前一行、后一行或留有至少两个空格紧跟在代码的后面，但是在代码与注释在同一行时不要使代码跟在注释的后面。另外，也不要将在注释放在代码中间。

【规则 3-1-82】对于常量、结构类型、函数/过程、自定义变量、代码段功能块、关键语句等有含义的代码部分，必须有注释行进行详细说明。

第2章　变量、常量与类型

【规则3-2-1】建议所有常量的第一个前缀必须为“C”，常量必须分主题归类定义，如有多个主题，每一个主题必须加一个主题前缀。前缀与前缀之间、前缀与名称之间用“_”分割。

正例

```
Const
    {主题1:plane}
    C_Plane_A    = 1;  // 含义
    C_Plane_B    = 2;  // 含义
      { 主题2 :bird}
    C_Bird_A     = 3;  // 含义
    C_Bird_B     = 4;  // 含义
```

【规则3-2-2】全局常量以“gC_”为前缀。

【规则3-2-3】每个常量的定义单独占一行，在同一行上，必须用“//”加注释说明该常量的含义。

说明　常量定义包括逻辑值的定义、常数值的定义、错误码的定义，其格式为：

```
Const      <常量> : 类型 = 值;      //注释
```

其含义是使用 Const 语句来声明用于代替某一数据值的常数。

正例

```
Const C_SQL:string ='select CH _USERER_ID from T_CD_USER'; //查询用户 ID
Const C_PI:real =3.1415926;  //圆周率
```

【规则3-2-4】按主题归类的常量，在每个主题开始的第一行必须用{}注释主题含义。

【规则3-2-5】结构类型定义时，类型名是以大写字母开头的字符串。

说明　结构类型定义格式：

```
Type
    <类型名>                                          //结构说明注释
            <变量名1>   :    变量类型;                  //变量说明注释
            <变量名2>   :    变量类型;                  //变量说明注释
    ...
    End  ;
```

正例

```
Type
TableList = record      //系统树单元类型定义
  InID : integer;           //单元编号
  Name : string;            //单元名称
  PID : string;             //上一级单元名称
  PT : PTableList;          //指向下一单元类型的指针
end;
```

【规则 3-2-6】在过程和函数中，当一个参数为记录型、数组类型 shortstring 或接口类型并且在例程中不被改变时，应将其定义为常量，以产生高效率的代码。

【规则 3-2-7】在变量区定义单元的全局变量。

【规则 3-2-8】每个需要注释的变量单独占一行，在行末用“//”注释其含义。

【规则 3-2-9】一个过程中的局部变量应遵循所有其他变量的使用和命名约定。临时变量也一样。

【规则 3-2-10】同一类型且含义逻辑上不并列的变量应分开定义，同一类型且含义逻辑上并列的变量应在一起定义。

【规则 3-2-11】不推荐使用全局变量。但是，在某些时候还必须使用，而且它们也只应在必须使用的时候才使用。在这种时候，应把全局变量限制在需要的环境内。

正例　一个全局变量可能只在单元的 Implementation 部分是全局的，所以必须把其定义在该部分内。被多个单元使用的全局数据应该被移入一个公共模块中。

【规则 3-2-12】全局变量可以在定义时直接初始化为某一个值。

说明　所有的全局数据会自动初始化为 0、nil、''、unassigned 等，称为零初始化。零初始化的全局数据在 EXE 文件中不占据任何空间，它被存储在一个虚拟的数据段中，待应用程序启动后再被分配到一段内存中。非零初始化的全局数据在硬盘的 EXE 文件占用空间。

【规则 3-2-13】在程序的开始阶段要对所有的变量赋予明确的数值。

说明　普通类型变量要选择合适的数值；类的实例如果此时不需要创建，那么就赋值为 NIL。所以，在主程序、过程或函数中要明确程序的初始化部分和程序的功能部分。

【规则 3-2-14】在必须使用全局变量时，尽可能在单元文件(.pas)的实现部分声明该变量。这样该变量就只在此文件中有效，而不会被其他文件访问。

说明　将全局变量放在单元文件的接口部分是很危险的。

【规则 3-2-15】一般用于例程内部的局部变量应在例程的入口处立即被初始化。

【规则 3-2-16】保留字的类型名称必须全部小写。

说明　Win32 API 类型通常全部大写,并且必须遵循在 windows.pas 或其他 API 单元中的详细类型名称的约定。对于其他变量名字,第一个字母应为大写,而其他字母应错落有致。

对于引用的其他公司、组织或个人的数据类型则尽可能地保持原样。

正例

```
var
MyString: string;                    // reserved word
WindowHandle: HWND;                  // Win32 API type
I: Integer;                          // type identifier introduced in System unit
```

【规则 3-2-17】对于浮点指针类型,尽量不使用 Real 类型,它只是为了和旧的 Pascal 代码兼容,应尽量使用 Double 类型。

说明　Double 类型是对处理器和数据总线做过最优化的、并且是 IEEE 定义的标准数据结构。当数值超出 Double 的范围时,可使用 Extended 类型。

【规则 3-2-18】对于自定义的数据类型,要以大写字母“T”为前缀,其他部分的命名遵守命名的一般惯例。

【规则 3-2-19】对于变体类型(Variant)和 OLE 变体类型,通常不建议使用。

说明　但在只有运行时刻才能知道数据类型的程序中必须使用该类型,这种情形多出现在 COM 和数据库开发中。OLE 变体类型使用在以 COM 为基础的编程中,例如自动化和 ActiveX 控制,而变体类型使用在非 COM 的编程中,这是因为变体类型可以十分有效地存储本地 Delphi 字符串(同一个字符串变量一样),但 OLE 变体类型会将所有的字符串转换为 OLE 字符串(wide-char 字符串)并且并不实例运算——它们永远拷贝。

【规则 3-2-20】不提供服务的数据类型定义格式为:Txxx =---------- ; // 类型含义(短横线表示没有类体的类)。

【规则 3-2-21】有状态并提供服务的数据类型遵循一定格式。

说明　有状态并提供服务的数据类型定义格式为:

```
Txxx = class(TObject)    //类型含义
private
  ...
protected
```

```
    ...
public
    ...
published
    ...
end;
```

【规则3-2-22】原则上，数据类型及其内部方法、属性、数据定义时应按字母顺序排列。属性的读写方法分别以“get”和“set”为前缀命名。

【规则3-2-23】所有内部数据放在 private 区。

【规则3-2-24】所有事件属性对应的方法指针放在 private 区。

【规则3-2-25】不准备被继承的属性的 get 与 set 方法放在 private 区。

【规则3-2-26】响应消息的方法放在 private 区。

【规则3-2-27】被子类调用的、但不能被外界调用的方法与属性，放在 protected 区。

【规则3-2-28】在 protected 区，有供子类重载的方法 virtual; virtual; abstract。

【规则3-2-29】在 public 区构建析构方法。

【规则3-2-30】供外界调用的方法放在 public 区。

【规则3-2-31】供外界调用的属性放在 public 区。

【规则3-2-32】出现在 Object Inspector 里供设计时用的属性，放在 published 区。

【规则3-2-33】出现在 Object Inspector 里供设计时用的事件响应，放在 published 区。

【规则3-2-34】特殊的数据应在定义行的行末用“//”注释其含义。特殊的属性、方法 应在定义行之前用“{}”注释其含义。事件指针的定义不需注释，但事件类型定义时必须在其前一行用“{}”注释其含义。

第3章　语　　句

【规则3-3-1】在程序体中,语句格式通常最多每10行语句有一个段落功能说明。

【规则3-3-2】begin和end语句要独占一行,begin要与上一层的第一个字母对齐,也就是说在换行之后不要留任何空格而直接写begin,end要与所对应的begin对齐。

【规则3-3-3】变量定义语句风格:var

var_name : <数据类型>;

【规则3-3-4】赋值语句风格:var_name: = <表达式>;

【规则3-3-5】条件语句风格:if 条件 then　//注释

```
begin
    语句;
    …
end
elseif 条件 then                    //注释
begin
    语句;
    …
end
else                                //注释
    语句;
```

说明　如果执行语句只有一句,可省略begin、end语句。

【规则3-3-6】if语句中,将最有可能执行的情况放在then语句中,不太可能的情况放在else子句中。这样对维护来说不会带来多少帮助,但会使程序效率更高。

【规则3-3-7】多级if语句的可读性不强,所以应经尽能避免出现多级if语句。

【规则3-3-8】不要嵌套5层以上的if语句。

【规则3-3-9】不要在if语句中使用不必要的括号。

说明 在源代码中,括号除了语法作用外就是在必要时对条件分段,以增加程序的可读性。所以在 if 语句中,如果条件明了而且在语法上不需要括号,那么就不要使用括号;如果条件过于复杂,那么就使用几个括号。总之,一切为了清晰。

【规则 3-3-10】如果在 if 语句中有多个条件要测试,应按照计算的复杂度进行排列。

正例 如有三个条件:Condition1、Condition2 和 Condition3。按照这个顺序,复杂度依次上升,与就是说 Condition1 比 Condition2 快,Condition2 比 Condition3 快,则 if 语句就这么写:

```
if Condition1 and Condition2 and Condition3 then
begin
   …
end ;
```

【规则 3-3-11】case 语句中,代表每种情况的常量应该按照数字或字母顺序排列。

【规则 3-3-12】case 开关语句风格:

```
//开关功能说明
case <表达式> of
   <值 1>:语句 1;
   <值 2>:语句 2;
   …
   <值 n>:语句 n
end;
```

【规则 3-3-13】每一个 case 情况的处理与语句应保持简单而不应该超过 4~5 行代码。如果所要执行的代码过于复杂,应采用独立的过程或函数。

【规则 3-3-14】case 语句的 else 子句只用于处理默认情况或进行错误检测。

正例

```
case i of
   1: begin
         …
      end;
   2: begin
         …
      end;
   else
      begin
         …
```

```
        end;
    end;
```

【规则 3 -3 -15】case 语句应遵循其他结构的缩进和命名约定。

【规则 3 -3 -16】for 循环语句风格：

```
//循环功能注释
for counter: = start to end do
begin
  ...
end;
```

【规则 3 -3 -17】while 循环语句风格：

```
//循环功能注释
while condition do
begin
  ...
end;
```

【规则 3 -3 -18】repeat 循环语句风格：

```
//循环功能注释
repeat
  ...
until condition
```

【规则 3 -3 -19】不要用 exit 过程来跳出一个 while 循环。如果需要的话，应该使用循环条件退出循环。

【规则 3 -3 -20】在一个 while 循环中所用的初始化代码应紧靠在进入 while 循环前面出现而不要被其他不相关的语句隔开。

【规则 3 -3 -21】如果在循环体中要考虑到意外处理，要搞清楚处理的方式和对相关变量的影响。

【规则 3 -3 -22】任何异常处理的结束部分(finally)应在循环之后立即进行。

【规则3-3-23】3种循环语句:for语句、while语句和repeat语句,如果循环次数确定,那么就使用for语句;如果在第一次循环之前要实现执行一次以获得初始比较数值,那么就使用repeat语句;其他情况基本上都可以使用while语句。

【规则3-3-24】repeat语句和while语句类似,且遵循同样的规则。

【规则3-3-25】在Delphi中,for语句中的循环变量不可以在循环体中被赋值。

说明 这是与Turbo Pascal不同的地方。

【规则3-3-26】在循环语句,尤其是多个循环嵌套使用的代码中,要小心使用break和exit关键字。

【规则3-3-27】with语句是一类非常容易出错的语句,使用with语句可以有效避免重复的输入工作,但是这样会使程序难以检查。所以,不要随便使用with语句。

【规则3-3-28】绝对不要使用带有两个或多于两个对象或记录的with语句。

说明 不要随便使用with语句的原因不是因为易于出错,而是因为难以排错。

反例

```
with Label1, Label2 do
begin
  ...
  Caption := "Delphi";
  ...
  end;
```

以上语句使人难以判断Caption是属于Label1还是Label2。

【规则3-3-29】with语句遵循本文档所说明的命名约定和缩进的格式规则。

【规则3-3-30】一旦资源被分配,必须使用try…finally来保证该资源能被正确地释放。

说明 异常的处理常被用于错误纠正和资源保护方面。

反例

```
someclass1 := TSomeClass.create;
someclass2 := TSomeClass.create;
try
  {…}
finally
   someclass1.free;
```

```
        someclass2. free;
    end;
```

正例

```
    someclass1 : = TSomeClass. create;
    try
    someclass2 : = TSomeClass. create;
    try
      { … }
    finally
      someclass2. free;
    end;
  finally
    someclass1. free;
  end;
```

【规则 3 - 3 - 31】除非是在单元的 initialization/finalization 部分或者对象的构造/析构中分配/释放资源，否则，凡是分配资源的地方都必须使用 try…finally 语句来保证资源得到释放。

【规则 3 - 3 - 32】不可以在 try…except 中使用 else 子句。

说明　因为这样有可能阻碍其他所有的异常发生。

【规则 3 - 3 - 33】需要在发生异常时执行一些自定义的任务，才使用 try…except，如果仅仅是为了显示一个错误信息，没有必要使用 try…except。

说明　因为 Application 对象能够自动根据上下文做到这一点。

【规则 3 - 3 - 34】如果要在 except 子句中激活默认的异常处理，可以用 raise 再次触发异常。

第4章　函数和过程

【规则3-4-1】事件、过程、函数应按主题分类归集在一起，每个主题开始的第一行用{}注释该类主题。

说明　约定归类方法如下：同一个组件的事件应归集在一起，被事件调用或与之相关的函数或过程应紧跟在该事件之后，函数、过程、方法之间如存在调用关系也应归集在一起，除非该函数、过程被多处调用。

【规则3-4-2】函数(Function)/过程(Procedure)编写代码时，须按一定风格。

说明

```
<Function/Procedure> Name (变量 :变量类型[,…]); //函数/过程说明
var
   变量1:变量类型;//变量说明
   变量2:变量类型;//变量说明
   …
begin
   语句;
   语句;
   …
end ;
```

变量声明、语句均从第3列写起，如有缩进，每次缩进两列，并与相对应的语句对齐。

【规则3-4-3】被多处调用的单元内部函数、过程应归集在单元的公共过程主题内。

【规则3-4-4】过程与函数的形参顺序主要应考虑寄存器调用规则，最常用的参数应当作为第一个参数，按使用频率依次从左到右排。

正例

```
procedure someproc(aplanet, acontinent, acountry, astate, acity);
```

【规则3-4-5】过程与函数的形参输入参数应当位于输出参数之前，并且范围大的参数应当放在范围小的参数之前。

【规则3-4-6】过程与函数中相同类型的形参应合并在一个语句中。

正例

```
procedure foo(param1, param2, param3 : integer; param4 : string);
```

【规则 3-4-7】过程与函数中参数的排序要按照日常惯用的顺序，如姓名、性别、年龄。

正例

```
procedure SetUserInformation(Name, ID: string; Age: integer);
```

【规则 3-4-8】所有的形参的命名要能够表达出该形参的用途。在合适的情况下，形参的名称最好以大写字母“A”为前缀。

【规则 3-4-9】Delphi 中有 const 和 var 形参，分别表示了对形参的读写控制。在形参列表中最好能够明确表示出来。

【规则 3-4-10】所有的常量参数最好标以 const 标记。

【规则 3-4-11】为避免命名冲突，除调用公共模块的例程外，调用其他模块的过程、函数时应当在例程名前加该例程所在的单元名。

【规则 3-4-12】除非确实需要在使用前再初始化，否则例程的内部变量应当在例程的入口处立即初始化。

【规则 3-4-13】代码中用到数据库字段名称时，统一用小写。

【规则 3-4-14】控件本身的属性、事件名，要按照 Delphi 自动提示的格式写。

【规则 3-4-15】过程与函数中一些非变参数也可用常量来传送。尽管这样做不会提高效率，但将给例程的调用者提供更多的信息。

第5章　类和类方法

【规则3-5-1】类的名字必须有意义并且类型的名字之前要加前缀“T”。

正例

```
type
TCustomer = class(TObject)
```

【规则3-5-2】类实例的名字通常是去掉“T”的类的名字。

正例

```
var
Customer: TCustomer;
```

【规则3-5-3】类的域名命名格式以“F”为前缀，表明这是一个域的名称。

说明　所有的域都必须是私有的，若想在类的范围之外存取域则必须借助属性。

【规则3-5-4】如果使用一个静态的类方法，那么该方法就不能被该类的后代类所继承。

【规则3-5-5】使用虚拟和动态的方法实现多重继承。

说明　在设计类的方法时，如果允许其后代可继承该方法，则应将其定义为虚拟的方法。只有在该方法有多个继承时（直接的或间接的）才使用动态的方法。例如，一个类类型包含一个可继承的方法，而100个后代类要继承这种方法，那么这个方法就会动态地产生100个后代类使用的内存。

【规则3-5-6】如果在一个类中使用抽象的方法，该类就不能被创建。只有在那些永远不会被创建的类中使用抽象的方法。

【规则3-5-7】所有存取类的方法都只能出现在类的 private 或 protected 部分。属性存取方法的命名应遵循过程和函数的约定规则。

【规则3-5-8】读取存取方法（方法读取器）必须以单词“Get”为前缀。写入存取方法（方法写入器）必须以单词“Set”为前缀。方法写入器的参数的名字应为 Value，并且它的类型应是它所操作的属性的类型。

正例

```
TSomeclass = class(TObject)
```

```
private
    FSomefield : integer;
protected
    function GetSomefield : integer;
    procedure SetSomefield(Value : integer);
public
    property Somefield : integer read Getsomefield write Setsomefield;
end;
```

【规则 3-5-9】对一个表示私有域的属性而言，应尽量少使用写入存取方法。

【规则 3-5-10】在一个单元文件中只可以定义一个类，而且单元文件的名称要与去掉前缀“T”的类名相同。如果在一个单元文件中定义了多个类，那么一个类就可以直接访问另一个类的私有字段或受保护字段，而不管该字段是否定义为只读。

说明　单元文件使用类的名称的目的是通过文件名可直接了解类的概况，这样便于使用。

【规则 3-5-11】类的外部接口通过方法实现，应当尽可能地使必要的方法作为接口，其他的方法要定义为私有方法。

说明　如定义方法 ExportNameList，其中又使用了一个子方法 ExportName，如果外部不会用到此方法，那么就将其定义为私有方法，将 ExportNameList 定义为公有方法。

【规则 3-5-12】在方法的实现代码中要注意字段的访问方法，可以直接使用字段来访问，也可以使用属性访问。

说明　如果要对字段进行写操作，那么就直接使用字段；如果仅仅进行读操作，那么就使用属性。其实，在任何条件下都使用字段也很值得考虑。

【规则 3-5-13】在程序编制完成后，如果要修改某个方法，那么尽量不要改变现有的接口而通过修改实现代码，或者提供新的接口方法来实现。

第6章 界面设计

在本规范中主要是为软件开发人员提供尽可能的界面设计指南,不可能为所有的界面提供建议。开发人员可以加以扩充,但是,应该尽量保持 Windows 程序的一致性。

制定本规范的目的是提高 Windows 应用程序之间及应用程序内部在视觉、功能上的一致性。这种一致性可以加快开发和学习的过程,提高生产率;减轻由于应用程序的界面不同而引起的混乱;给用户一种稳定感。

【规则 3-6-1】菜单的类型分为下拉式菜单、弹出式菜单和级联菜单。

【规则 3-6-2】在下拉式的菜单中,应该具备菜单标题和菜单项。菜单标题在菜单栏上。

【规则 3-6-3】菜单的标题不能包含数字或空格,菜单的标题应由中文标题和一个通过键盘直接访问菜单的带下画线的助记符(即快捷键)组成。

说明 助记符的选择原则为(以下按优先级排序):菜单名称的英文单词的第一个字母,标题中的一个有特色的辅音字母,标题中的一个元音字母。菜单的标题不允许用相同的助记符。

【规则 3-6-4】菜单项的名称应该惟一,菜单项的名称后应该有一个助记符,助记符的选择原则同上。

【规则 3-6-5】如果菜单项可以按逻辑分组,各组之间应该用一条线分开。

【规则 3-6-6】菜单中的快捷键应该在菜单项的后面。

【规则 3-6-7】如果菜单命令显示一个对话框,在菜单的标题中需要有标题提示“...”,如“打开(F)...”。

【规则 3-6-8】在弹出式菜单中,设计原则和下拉式菜单中的原则相同,不同的是在菜单项中不能包含快捷键。

【规则 3-6-9】在级联式菜单中,设计原则和下拉式菜单中的原则相同,不同的是在菜单的级联的菜单层数不能超过 3 层。

【规则 3-6-10】消息对话框是用来显示错误消息和其他重要信息的模式对话框。在对话框中包含标题栏,用来表示消息的来源。信息框统一采用 MessageBox。

说明　窗体/对话框设计是界面设计的重要部分之一。在消息对话框中还包含一个显示消息种类的图形符号，消息的 3 种类型分别是信息消息、警告消息、极重要消息。

【规则 3-6-11】在窗体的设计中，建议窗体中的颜色设置为标准的 Windows 颜色，以灰色为主。窗体中文字颜色为黑色。

【规则 3-6-12】窗体中的字体为宋体，字号为 5 号。

【规则 3-6-13】窗体的大小应为 60 的倍数。

【规则 3-6-14】窗体中的标题栏中不能含有图标，文字对齐方式为左对齐，窗体中要包含关闭按钮。

说明　如果需要，可设置显示为“?”的帮助按钮。

【规则 3-6-15】窗体中主要控制项要包含快捷键。

【规则 3-6-16】窗体的位置，通常水平和垂直地置于应用程序窗口的中央。

【规则 3-6-17】如果按钮命令显示一个对话框，则在显示的按钮命令中应有“...”提示。

【规则 3-6-18】按钮的大小，高为 27，宽为 90（在将按钮和对话框的网格对齐时，网格的间隔设为 10～15）。

说明　当按钮的标题超过两个汉字时，可以适当增加按钮的宽度。如果需要，可以增加按钮的高度，当改变按钮的形状及风格时，应该经过项目组的同意。

【规则 3-6-19】每个对话框都应至少包含一个关闭对话框的按钮。只要求用户确认（而不用选择）的消息对话框只包含一个标以“确认”的按钮。其他所有对话框都至少包含两个按钮：一个关闭对话框并启动一个动作；另一个关闭对话框但并不启动任何动作。

说明　启动动作的按钮通常标以“确认”，关闭对话框但不启动任何动作的按钮通常标以“取消”。某些对话框（称为多动作对话框）另外包括一些允许用户启动某些动作但不关闭原对话框的按钮。在这些对话框中，如果由这些附加按钮执行的动作无可挽回地修改了用户的数据，则应将“取消”按钮的标签改为“关闭”。不管这个按钮标以“取消”还是“关闭”，用户都可以通过“Esc”键表示按下这个按钮。如果标签是“关闭”，还应该用带下画线的字母“C”作为其助记符。

【规则 3-6-20】对话框中的一个命令按钮可以被指定为默认按钮，它在用户按“Enter”键时表示选择了该命令，默认按钮显示一个粗边框以区别于其他按钮。

【规则 3 -6 -21】按钮的动态标签是指按钮的标签可以根据两种方式改变当前用户的可用性能:如果按钮代表的动作当前不可用,则其标签应当变暗;如果按钮代表的动作的性质根据环境产生改变,则其标签也会相应地变化。

【规则 3 -6 -22】只要有可能,建议按钮的排放遵循一定的放置规则。第一种是从右上角起沿窗体的右边界排放。

说明　在这种方式下,按钮通常是等宽的。根据命令按钮是否启动一个动作将它们编组。如果存在一个“确认”按钮,它应和“取消”按钮要编在一个组里,和其他动作按钮分开。如果不存在“确认”按钮,“取消”按钮可以和其他动作按钮编在一起。在一按钮的底部和同组中下一个按钮的顶部之间相隔距离为 10。组与组之间的间隔为 15。在对话框的边界和按钮的边界(指第一个按钮的顶部,最后一个按钮的底部,以及所有按钮的右边界)之间间隔为 20。

【规则 3 -6 -23】建议按钮的排放的第二种规则是沿窗体底部排成行。

说明　根据命令按钮是否启动一个动作将它们编组。在一按钮的右边界和同组中下一个按钮的左边界之间相隔 10。组与组之间的间隔为 20。在窗体的边界和按钮的边界(指第一个按钮的左边界,最后一个按钮的右边界,以及所有按钮的底部)之间间隔为 20。正常情况下按钮应该是等宽的,当个别按钮可以宽一点以容纳过长文字。

【规则 3 -6 -24】典型的按钮排放顺序是默认按钮放在顶部或者在左边,其后是其他可启动动作的按钮,再是“帮助”按钮。

【规则 3 -6 -25】如果仅使用一种方式没有足够的空间容纳下所有的按钮,可以用规则 3 -6 -22排放最重要的命令按钮,而用规则 3 -6 -23 排放其他所有的命令按钮。

说明　图 3.6.1 表示了一种按钮的排放方法及其间隔大小,其中矩形代表窗体中的按钮。

【规则 3 -6 -26】窗体中 Text 设置:Text 框的高度与 ComboBox 的缺省高度一致,Text 框之间的间隔为 15。

【规则 3 -6 -27】窗体中 CheckBox 设置:CheckBox 的高度与 ComboBox 的缺省高度一致,CheckBox 之间的间隔为 15。

【规则 3 -6 -28】窗体中 Option 设置:Option 的高度与 ComboBox 的缺省高度一致,Option 之间的间隔为 15。

【规则 3 -6 -29】窗体中 Label 设置:Label 的高度与 ComboBox 的缺省高度一致,Label 之间的间隔为 15。

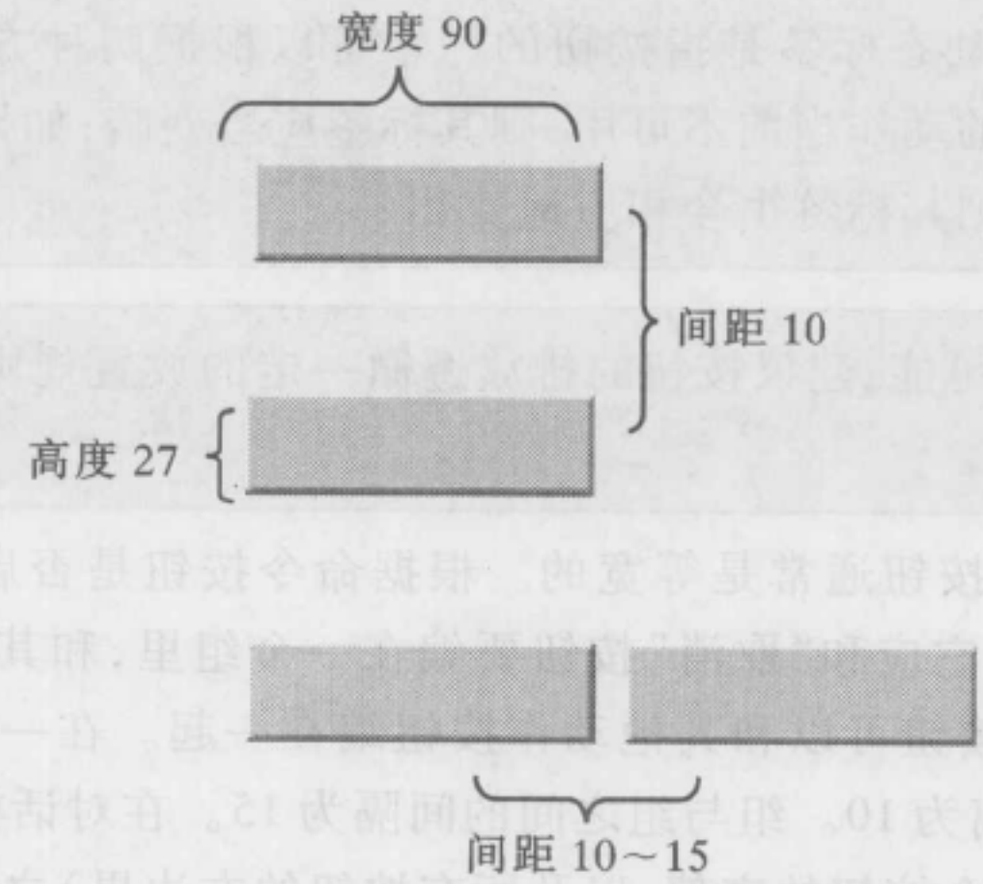


图 3.6.1　按钮排放方法及间距

【规则 3－6－30】窗体中 TabPage 设置：如果采用 Ms 的 SSTab 控件或者类似的 Tab 控件，则必须将该控件设置为属性页的显示方式。

【规则 3－6－31】在窗体中的控件和窗体、控件和控件之间的位置关系中，窗体中的控件（已经做了规定的除外）与窗体对齐边缘的距离为 15。

【规则 3－6－32】应将“帮助”按钮放在其他按钮之后，使它位于对话框的右下部附近。

第7章　其他规范

7.1　防错误处理

【规则3-7-1】在涉及一些系统性的操作时，必须使用防错误的语句。

说明　这些操作包括：数据库连接和数据库访问、网络访问、文件存取、对象句柄的使用（如使用窗口对象的句柄等）、指针使用等。

【规则3-7-2】防错误的语句包括以下几种情况：第一是一般合法性判断，使用 assert，但 Delphi 不支持；第二是异常处理，使用开发工具或语言提供的错误处理语句（如 Delphi 中 try ... except ... end 或 try ... finally ... end 语句块等）。

7.2　COM　接　口

【规则3-7-3】使用基于 COM 技术的接口方式可以采用 ActiveX EXE、ActiveX DLL 等形式实现。

说明　建议采用 ActiveX DLL 形式提供工具的应用逻辑层，在 ActiveX DLL 中可以提供适当的 Form 来作为界面。

【规则3-7-4】对于各个工具提供的 ActiveX DLL，应该是一个可以通过外部创建的对象。即在 ActiveX DLL 中的 Class 属性应该为 MutiUse 或者 GlobalMutiUse。在这个外部可以创建的对象中，提供了和框架的接口，同时提供了显示工具窗口的方法。

说明　如得到设计对象标识码的方法和显示可靠性预计的窗口的方法。

【规则3-7-5】注意在 ActiveX DLL 中提供一个 MDI 的子窗口可能会有问题。因为 ActiveX DLL 是有限制的。

7.3　窗体和包

【规则3-7-6】对于自动创建窗体，只有主窗体可以是自动创建的。

说明　所有其他的窗体都必须从工程选项对话框中的自动创建列表中移走。

【规则 3-7-7】所有的窗体单元都应包含一个窗体实例化函数，该函数用来创建、设置、模式地显示窗体，并释放窗体。该函数应返回窗体的模式结果。

说明　该函数要传递的参数应遵循本文档指定的“参数传递”标准。通过这种方式封装的函数有助于代码的再利用和维护。该窗体的变量要从单元中移走，并在窗体实例的函数中进行本地式地定义。注意，这就意味着该窗体必须从工程选项对话框中的自动创建列表中剔除。

【规则 3-7-8】自定义组件在运行包和设计包的使用上是不同的，运行时刻的包应只包含其他控件包所要求的单元或构件。包含属性/控件编辑器和其他只为设计用的代码应放入到设计时刻包中。注册单元应放在设计包中。

7.4　修改代码

【规则 3-7-9】在修改代码规范中，要求保留修改前的内容、标识出修改和新增的内容。并在文件头加入修改人、修改日期、修改原因及修改说明等必要的修改历史信息。

【规则 3-7-10】对于新增代码行，应在其前后添加注释行说明。

正例

```
// 修改人，修改时间，修改说明
新增代码行
// 修改结束
```

【规则 3-7-11】同样对于删除代码行，也应在其前后用注释行说明。

正例

```
//修改人，修改时间，修改说明
//要删除的代码行（将要删除的语句进行注释）
//修改结束
```

【规则 3-7-12】修改代码行的方法是先删除代码行，然后再新增代码行。

正例

```
//修改人，修改时间，修改说明
//修改前的代码行
//修改结束
//修改后的代码行
修改后的代码行
//修改结束
```

第四部分

VC ++ 编程规范

第1章　编程环境设置

【规则 4 - 1 - 1】为了减少代码的冗余、提高运行速度，应删除 Stdafx. h 与 Stdafx. cpp 文件。

说明　在 VC ++ 中，用向导（Wzard）生成一个工程（Project）时，如 Win32 Application 等，如果不是选择生成“An Empty Project”，那么在工程中总是包含 StdAfx. h 和 StdAfx. cpp 文件。应该把 StdAfx. h 文件中所含的用户需要的头文件直接包含（Include）在 CPP 文件中，然后删除掉这两个文件。

本规范规定：CPP 文件应只明确地包含用户所需的头文件，而不是把所有的头文件都包含在 Stdafx. h 文件中。但从工程中删除掉这两个文件后，会产生编译错误，因此必须改变编译环境的设置选项，具体方法是选择“Project→Settings→C/C ++”，在其中的“Category”下拉列表中选择“Precompiled Header”，并单击选中“Not using precompiled headers”项，如图 4 - 1 - 1 所示。

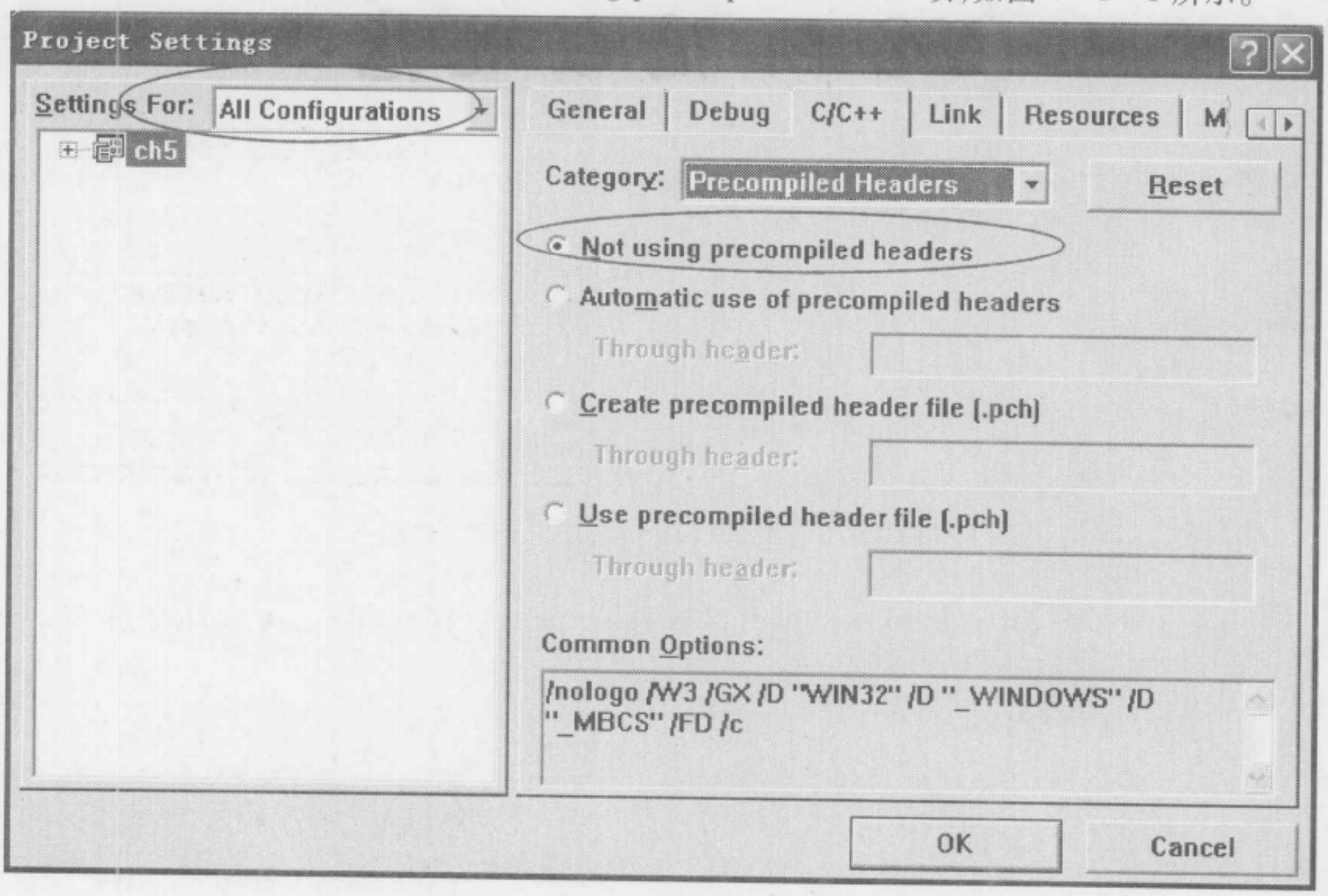


图 4 - 1 - 1　项目设置中的 C/C ++ 选项卡

【规则 4 -1 -2】在开发商用软件的任何一个程序时,都不要使用 Debug 进行编译调试,一定要用 Release 进行工作,否则后患无穷。通过设置可使 Release 版的工程(Project)含有 Debug 信息。

说明　VC ++默认生成的工程没有调试信息,即不能进行源码级的调试。欲使工程中包含 Debug 信息,应选择“Project→Settings”,在出现的项目设置对话框中,按图 4 -1 -2 所示设置“C/C ++”选项卡内容,按图 4 -1 -3 所示设置“Link”选项卡内容。

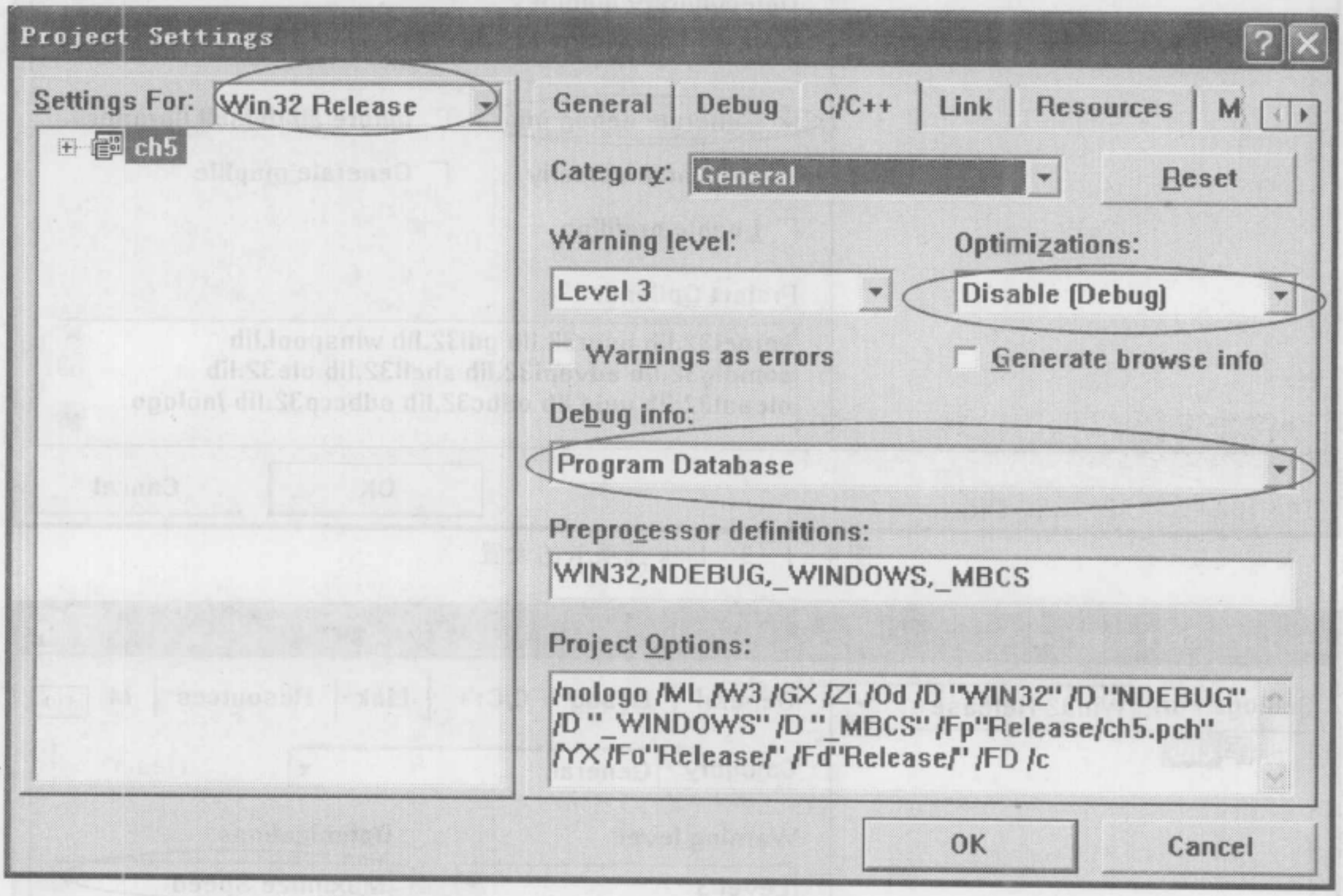


图 4 -1 -2　C/C ++选项卡的设置

但应注意,在工程完成后需要打包提交时,应该把这些调试信息删除。这时,可按图4 -1 -4 及图 4 -1 -5 所示重新设置“C/C ++”及“Link”选项卡的内容。

【规则 4 -1 -3】为了阅读代码方便,建议将 Tab 设为 8 个字符的宽度。

说明　VC ++中,Tab 的默认设置是 4 个字符宽,这样使得代码得缩进不明显,造成代码阅读困难。本规范中将 Tab 设为 8 字符宽。Tab 在传统的文本中的值也是为 8,这样使代码与传统的文本一致。

具体的设置方法是选择“Tools→Options→Tabs”,如图 4 -1 -6 所示。

【规则 4 -1 -4】建议合理地设置开发环境的颜色。

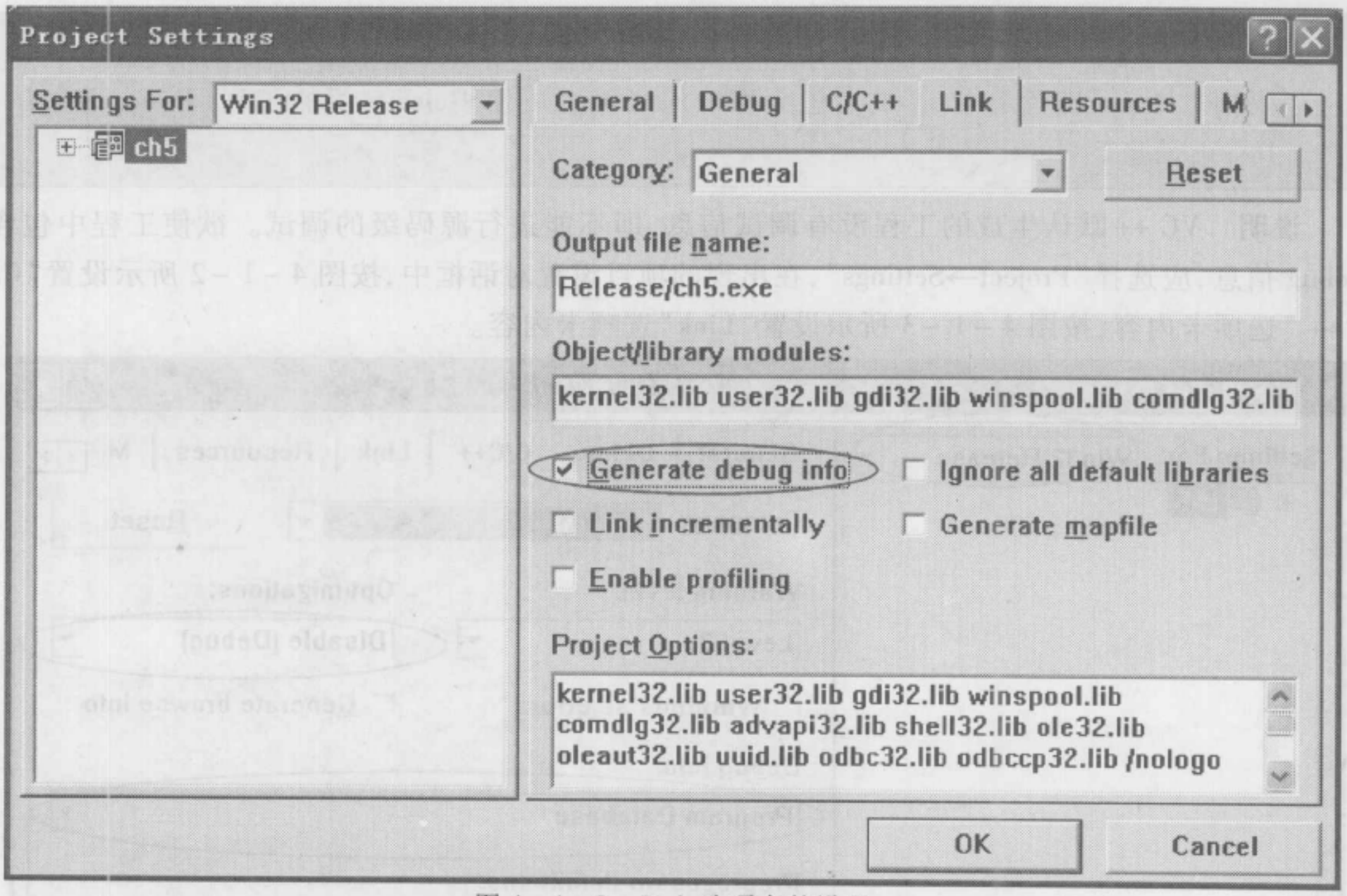


图 4 -1 -3　Link 选项卡的设置

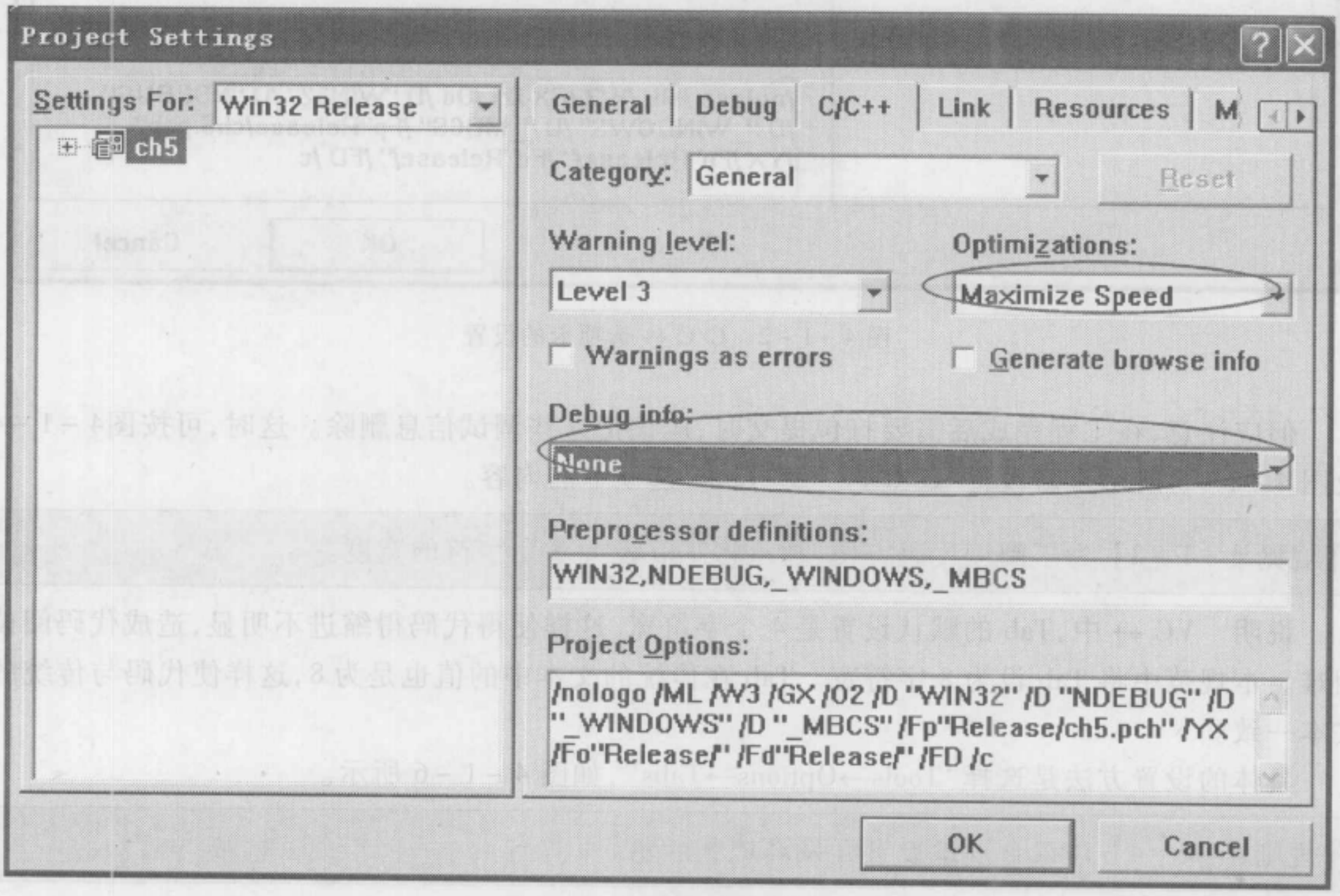


图 4 -1 -4　删除调试信息

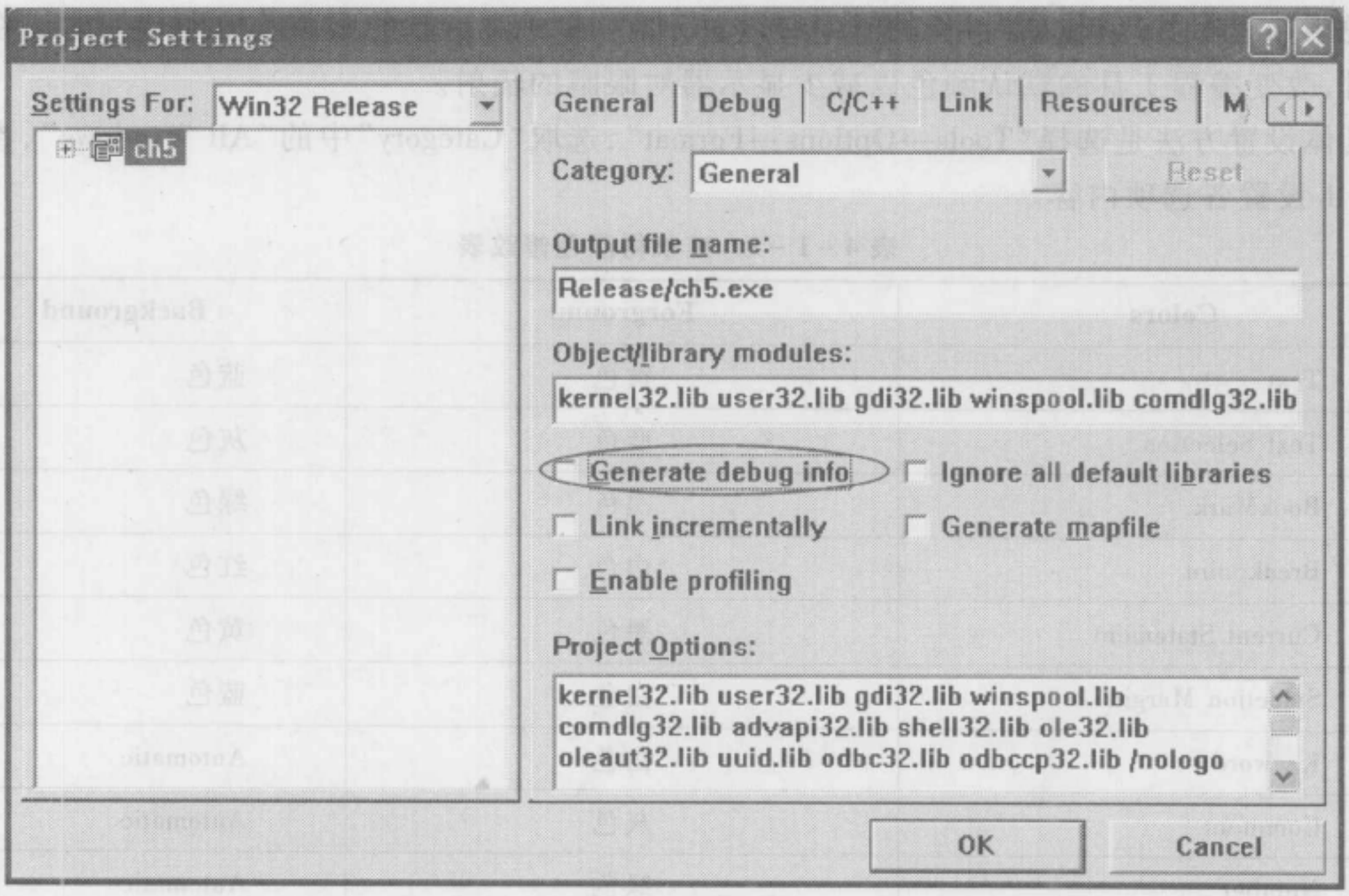


图 4－1－5　删除调试信息

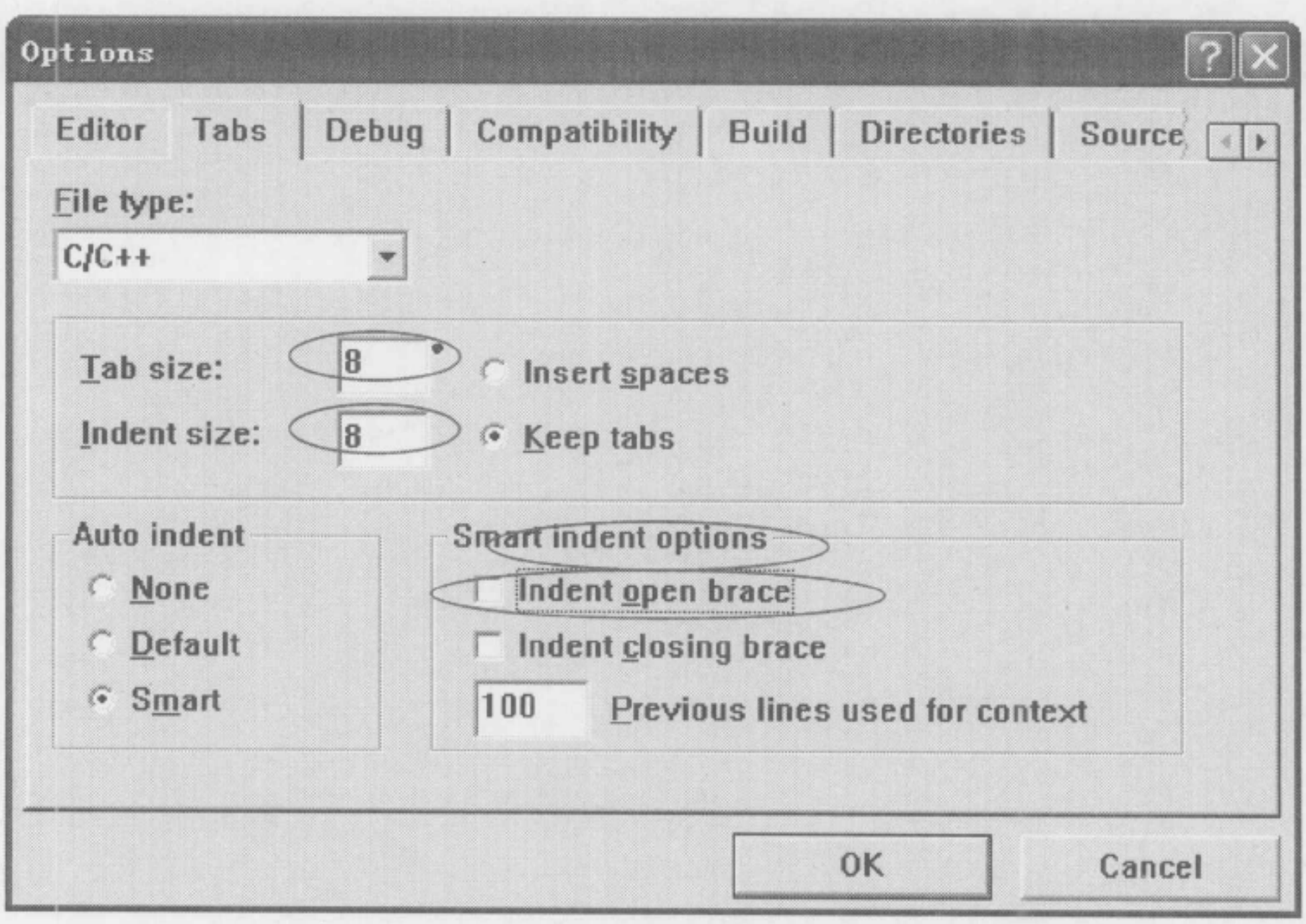


图 4－1－6　Tab 值的设置

说明　程序员每天总有很长的时间盯着显示器，这样对自己的眼睛伤害很大。建议在编写程序时，改变编程工具的默认颜色以减少显示器对眼睛的辐射。

具体设置方法是选择“Tools→Options→Format”，选取“Category”中的“All Windows”，参照表 4 -1 -1 设置各选项内容。

表 4 -1 -1　建议得颜色修改表

Colors	Forground	Background
Text	黄色	蓝色
Text Selection	蓝色	灰色
BookMark	黑色	绿色
Breakpoint	白色	红色
Current Statement	黑色	黄色
Selection Margin	白色	蓝色
Keyword	白色	Automatic
Comment	灰色	Automatic
Number	绿色	Automatic

第2章　布局及变量

【规则4-2-1】按规定“{”、“}”必须对齐。

说明　“{”、“}”之间的内容表示一个块(Block),是一个相对完全的语义单元。其在垂直方向上必须对齐,以使得各语义块层次清楚,语义的递进关系明确。

【规则4-2-2】函数体中最外层的“{”、“}”是语义范围最大的一对“{”、“}”,它应和函数的最左面对齐。

反例

```
void  Test( )
|→Tab←| {
|→Tab←| }
```

正例

```
void  Test( )
{
}
```

【规则4-2-3】在函数体内的块,若“{”在一行的开始处,则必须和其上一条语句在垂直方向上向右缩进一个Tab值。

正例

```
if( a = = b&&c = = d)
|→Tab←| {
...
|→Tab←| }
```

【规则4-2-4】若一行语句较短,用到“{”时可不换行。

反例

```
while (1)      //语句较短的情况
|→Tab←| {
...
|→Tab←| }
```

由于while(1)语句较短,使下一个语义块(“{”,“}”)有种悬空的感觉,代码看起来松散、无序。

又如以下的情形:

```
if(bFlag)
|→Tab←| {
…
  // 里面有一大段代码
|→Tab←| }
```

正例

```
while(1){
|→Tab←| …
|→Tab←| }

if(bFlag){
|→Tab←|…          // 里面有一大段代码
|→Tab←| }
```

改为如此的换行方式时,代码看起来则紧凑、面貌一新。

【规则 4 -2 -5】当一行语句较长时,情形与上一条正好相反,用到“{”时必须换行。

说明　当一行语句较长时,“{”被“隐藏”在一行较长的语句后面,使块的语义层次较模糊。

反例

```
if(a = =b&&c = =d&&e = =f…){
    …
                                    }
```

正例　应该改为:

```
if(a = =b&&c = =d&&e = =f…)
|→Tab←| {
|→Tab←|…
|→Tab←| }
```

一般当一行宽度超过 8 个字符时应将“{”放在下一行。

【规则 4 -2 -6】VC ++ 中有一个自动的格式整理功能,只需选取需要的代码,按 Alt + F8 就能够自动地将代码整理成微软的 CPP 格式了。再按以前的规范简单地整理一下,代码就很好使用了。

反例　有如下的一段代码:

```
    int   m;
for (m =1; m <10; m ++)
{
    printf ("'it's a Test!    ");
        printf ("m =%d   ",m );
}
```

可以看出,以上的代码很乱,直接对它进行整理可能要花很多时间。现在,只要选取以上代码段按 Alt + F8 或运行“Edit→Advanced→Format Selection”项(如图 4 - 2 - 1 所示),就能得到如下比较规范的代码段:

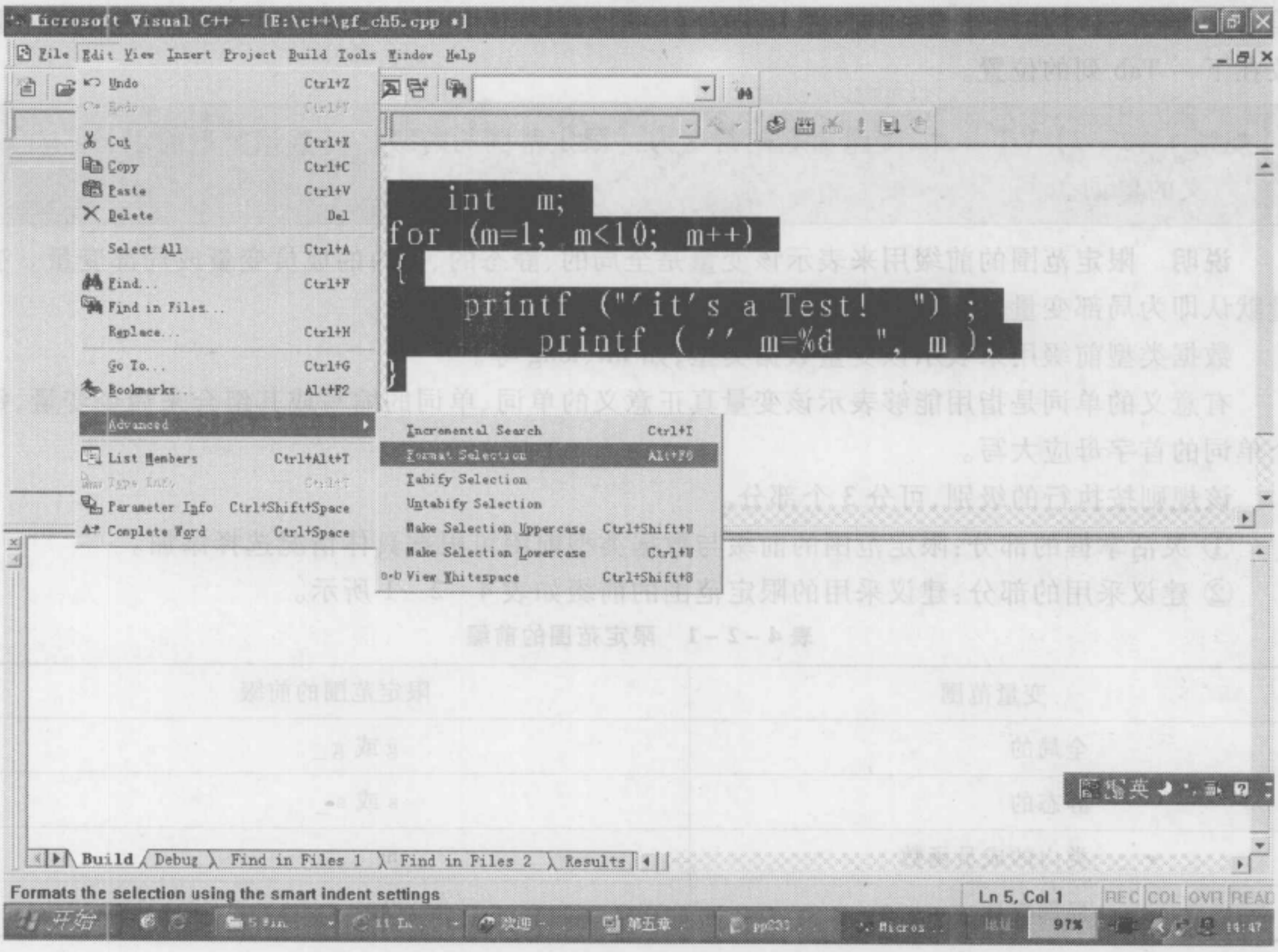


图 4 - 2 - 1　选取代码段及快速的代码整理方法

```
int   m;
for (m = 1; m < 10; m ++)
{
    printf ("' it's a Test!    ") ;
    printf ("m = % d   ",m);
}
```

正例　现在可以再对以上代码进行整理,可变成如下的格式:

```
int   m;
for (m = 1;m < 10;m ++)
    {
    printf ("' it's a Test!    ") ;
    printf ("m = % d   ", m);
```

}

此例中的代码并未直接用微软的规范格式，这是因为微软的规范格式没有把一对花括号利用起来。例如，在 if 语句中，微软的格式是“{”直接在 if 的下一行语句的对应列位置，这样读代码时，就会受这对花括号的影响。因为花括号中的内容实际是下一个层次的内容，所以应该将其放在下一 Tab 列的位置。

【规则 4 -2 -7】VC ++ 中，变量命名的约定为：[限定范围的前缀] + [数据类型前缀] + 有意义的单词。

说明　限定范围的前缀用来表示该变量是全局的、静态的、类内的成员变量或局部变量。变量默认即为局部变量，故无须任何限定范围的前缀。

数据类型前缀用来表示该变量数据类型，如 int、long 等。

有意义的单词是指用能够表示该变量真正意义的单词、单词的缩写或其组合来命令变量，每个单词的首字母应大写。

该规则按执行的级别，可分 3 个部分。

① 灵活掌握的部分：限定范围的前缀与数据类型前缀可根据具体情况选择添加。

② 建议采用的部分：建议采用的限定范围的前缀如表 4 -2 -1 所示。

表 4 -2 -1　限定范围的前缀

变量范围	限定范围的前缀
全局的	g 或 g_
静态的	s 或 s_
类内的成员函数	m
局部变量	无

数据类型前缀的命名方法建议采用匈牙利命名法。

③ 必须执行的部分：即“有意义的单词”部分。一般常用变量命名方式如下：

- 循环变量可以直接定义成 i、j、k、l 等单一字母变量，大、小写不限。
- 计算变量可定义为 Cnt，Count，Counter 等。
- 返回值变量可定义为 Ret。
- 位置变量可定义为 xPOS，yPOS 等。
- 用 Width、Height、Wide 等表示具有宽、高含义的变量。

【规则 4 -2 -8】在 C ++/VC ++ 中，务必注意变量定义的地方，正确应用变量。变量的定义应出现在与之相关的所有处理代码之前。

说明　在 C 语言中，变量必须定义在使用它的代码之前，否则编译无法通过。而在 C ++ 中，变量的定义可以和代码交替进行，即所谓的“随用随定义”的自由风格。本规范规定：变量定义不一定定义在函数块的变量定义区，但必须在它所在块的变量定义区，即在它的所有处理代码之前定义。

反例　以下为错误的使用方法：

```
main()
    {// 块的开始,变量应该在此处定义
     // 变量定义区
     int   i = 1;
     printf("print i = %d\n",i);
     //此变量应该在块的开头,即在变量定义区内定义
     int   j = 2;
     printf("print j = %d\n",j);
    }
```

上面的代码书写方式造成了块中的变量和代码混在了一起。

正例　正确的方法应该是把变量定义在变量区，把代码处理写在代码处理区，从而使块中的变量和代码分离（实际上，编译器就是这么做的）。如下为正确的使用方法。

```
函数名
{// 块的开始
    // 在块的开始处定义变量
    //变量定义区:
int   i = 1;

//代码处理区:
printf("print i = %d\n",i);
for 循环
    {//块的开始

     //在块 BLOCK)的开始处定义变量
     //变量定义区:

     int j;
     …
     //代码处理区:
     printf("print j = %d\n",j);
     …
     }//块的结束
}//块的结束
```

【规则 4-2-9】变量定义时，应遵循如下对齐原则：变量数据类型 + *N* 个 Tab + 变量名 + [*N* 个 Tab] + “ = ” + [初始化值] + “ ; ”。

说明　其中,变量数据类型与变量名之间的“*N* 个 Tab”中的 *N* 为直到所有变量名垂直对齐为止的值,其值必须大于 l。后面的“*N* 个 Tab”中的 *N* 的值可等于 0,但建议也使后面的初始化值按 Tab 值垂直对齐。

正例

```
INT       nFullPicWidth    =0;
INT       nFullPicHeight   =0;
LPINT     lpwidthIndex     =NULL;
LPINT     lpHeightIndex    =NULL;
LPDIB     lpOriginDib      =NULL;
LPDIB     lpScaleDib       =NULL;
RECT      rcViewRect       ={0,0,0,0};
POINT     posOrigin        ={0,0};
POINT     AnchorPt         ={0,0};
SCALERECT   rcScale;
int x,y;
int X,Y;
int i;
```

另外,变量的上下排列顺序按照上宽下窄的倒三角排列方式进行排列,同时考虑到变量的长短可能相差较大,或参数的个数较多的情况,也可以考虑“分块,块间换行,块内对齐”方法或参照“定义的先后顺序”进行对齐。

第3章　头文件、注释及其他

【规则 4 -3 -1】在头文件中,必须包含防止重复编译的预编译指令。

说明　为了使头文件不被重复包含(这样会引起编译错误),需要加入以下预编译指令,假设头文件名为:student. h,则可以加入如下预编译指令。

```
# ifndef   STUDENT _ H
# define   STUDENT _ H
… // 具体申明
#endif
```

建议上面的字母 STUDENT _ H 都用大写,且下画线不能漏,都用小写也不会出错。

【规则 4 -3 -2】用 extern 引出的变量,必须说明该变量的出处。

说明　出处可以写在该变量的上面(当该变量上有注释时)或后边。如下所示:

```
// 来自 abc. cpp
extern   BOOL   g _ bThreadExit;          // 线程是否退出的开关量
```

【规则 4 -3 -3】在 VC ++ 中,注释(特别是函数注释)应当更加详尽。

说明　注释对于程序犹如眼睛对于人的重要性一样,没有注释的程序对于读者来说好比眼前一团漆黑,而不规范的注释和好几千度的近视眼也没有什么区别。

函数的注释如下所示:

```
////////////////////////////////////////////////////////////////////////
//
//函数名:
//        BOOL HandleWriteData(LPOVERLAPPED,LPCSTR,DWORD)
//
//目的:
//        向一个 COM 文件句柄(FILE HANDLE)写入一个字符串
//
//参数:
//        lpoverlappedWrite:在 WriteFile 函数中需要用到的重叠结构
//        lpszStrjnqToWrite:待写入的串
//        dwNumberOfBytesTowrite:待写入的串的长度
//
//返回值:
```

```
//        如果所有的字节都被成功写入则返回 TRUE。如果非整个串被成功写入则返
//        回 FALSE。
//
//功能:
//        此函数为写线程(Write Thread)服务的一个帮助(Helper)函数。它实际上
//        是写入一个串到 comm file。注意此函数将会是阻塞式的,它一直等待写入
//        完成或 CloseEvent 去通知(singal)线程结束。另外一个返回 FALSE 的情况是
//        comm  port 本身即是关闭的。
//
////////////////////////////////////////////////////////////////////////////
```

【规则 4-3-4】一块语句间最好用“{”、“}”标识出来。

说明　每个单独的语义最好使用花括号对,哪怕是一行语句。

反例

```
if( a = =b&&c = =d&&e = =f)
|→Tab←|  return  FALSE;
```

正例

```
if( a = =b&&c = =d&&e = =f)
|→Tab←| {
            return  FALSE;
|→Tab←| }
```

【规则 4-3-5】当判断条件较多时,要换行对齐。

说明　当判断条件较多,并且条件本身在意义上就有平行的条件和非平行的条件,如下所示:

```
if( a = =b&&c = =d&&e! =f &&g! =h || i = =j &&k = =l && m! =n && o! =p)
{
...
}
```

由于判断条件较多,且条件本身有平等的和递进的条件之分,故代码可以写成如下方式:

```
if( a = =b && c = =d
&& e! =f && g! =h
|| i = =j && k = =l
&& m! =n && o! =p)
|→Tab←| {
|→Tab←| …
|→Tab←| }
```

注意:条件与条件分隔符之间最好有个空格。

【规则4-3-6】内部使用的函数可以采用_stdcall、_cedel、static,但给外部使用的函数建议采用APIENTRY、APICALL。

正例 在CpuType.h中:

```
#define  APICALL  _stdcall
```

【规则4-3-7】定义变量时的变量类型的书写建议参照CpuType.h。

说明 即变量类型必须使用宏方式来说明,以便将来代码能移植到IA-64之上。不管是16位、32位、64位,还是未来的CPU,其中的CHAR、BYTE、WORD、DWORD、QWORD均是不变的,但INT类型又有int8、int16、int32、int64等之分,LONG可能是long32、long64等。所以,针对不同的CPU的系统,应该使用不同的数据类型。

Windows的其他类型,如LPSTR和LPBYTE、HANDLE、LPVOID、HMODULE、HFILE等按Windows的方式说明。

【规则4-3-8】最好使函数的功能单一化,尽量避免巨型函数。但也不能走向极端,写出一堆无意义的函数。

【规则4-3-9】建议每个工程都维护一个Readme.txt文件,主要用来记录每次重大功能的添加、修改。

说明 Readme.txt文件的具体格式为:

```
日期 |→Tab←| 修改的文件名 |→Tab←| 修改的功能 |→Enter←|
修改后的功能、原理说明:
  …

日期 |→Tab←| 修改的文件名 |→Tab←| 修改的功能 |→Enter←|
修改后的功能、原理说明:
  …
```

【规则4-3-10】注释可长可短,但应该是画龙点睛、简单扼要的。需要的地方必须加上,特别是在语义转折处。

【规则4-3-11】复杂函数的注释必须严格按照上述规则4-3-3执行,简单的函数可以用一句话简单地注释在其上面。

正例

```
// 求两个值的最大值
void Max(int &x,int &y)
```

【规则 4 -3 -12】内部使用的函数可以简单地注释，但供别人使用的函数必须严格注释，特别是入口参数和出口参数。

第4章　优化编码

【规则4-4-1】内存分配、释放编程时，要养成成对编码的习惯，即内存的分配与释放成对进行。

说明　在C/C++中，内存的分配、释放由程序员自己管理。如果不能很好地解决内存的分配、释放的问题，就很容易引起程序的内存泄漏或内存非法访问。对该问题较好的处理方法是在编程时养成成对编码的习惯，即内存的分配与释放成对进行 。在编写了内存的分配代码时，再接着写个相应的内存释放的代码，而不是等到了最后，有了一大堆的内存的分配，再去写内存释放的代码，这种最后再补漏的做法往往不能达到补漏的作用，造成了程序的不稳定性。

【规则4-4-2】成对编码，简单地说，就是输入完一个“{”后，接着输入“}”，这样，这一对大括号就被同时写入程序中，当写完分配内存函数后，立即写释放函数，再添加中间代码部分。

正例

第一步：写分配内存函数。

```
lpBuffer = LocalAlloc( LPTR, 512) ;
```

第二步：写释放内存函数。

```
lpBuffer = LocalAlloc( LPTR, 512) ;

LocalFree( lpBuffer) ;
```

第三步：在中间空格部分添加代码。

```
lpBuffer = LocalAlloc( LPTR, 512) ;
for ( i = 0; i < 256; i ++ )
    {                    //第一层代码
    }
LocalFree( lpBuffer) ;
```

第四步：在“{”、“}”中间添加代码。

【规则4-4-3】应充分地应用代码缩进形式，使代码的层次关系更清楚。

说明　输入一条语句后，如“for(i = 0; i < 256; i ++)”，接着按Enter键，按一下Tab键，使输入的“{”纵列一定要与“for”相差一个Tab。当一段代码很大的时候，移动键盘的上下方向键时，光标就会自动地滚动到对应的层次上。

这样，编写出的代码就能使人看到清楚的层次。比如，从下面的程序段中可以看出if和else在一个层次上。当程序比较大的时候，就可以清楚地知道层次的关系，可以选取自己感兴趣的段

进行阅读。例如,在一个很大的算法中就可以把算法切成块,分块地进行消化。下面是一段演示代码:

```
lpBuffer = LocalAlloc( LPTR, 512);
for (i = 0; i < 256; i ++ )
    {                    //第一层代码
   if (i = = 65)
        {                    // 第二层代码
         // 代码 1
         }
    else{
         // 代码 2
         }
    }
LocalFree( lpBuffer);
```

其实,这样写成的代码就相当于一个树的结构,for 接下的一对大括号是“根”,而 if 是代码 1 的“根”,else 是代码 2 的“根”。代码这种一层一层的结构,就形成了一个类似树的结构。

编写程序的过程就是一个生成树的过程,可以先有主干的根,接着就可生成其对应的子树,子树又可以生成下一级的子树。当阅读代码时,只需要阅读对自己有用的一块或者说是一个子树的部分。

【规则 4 -4 -4】当有一段代码需要理解时,可以不必急急忙忙就去从头到尾进行阅读,最好先把它们按以上的规则进行规范。

说明　读代码的时候分块去进行阅读,而不是从头读到尾。成对编码的好处是使得随时能运行正在编写中的程序。

【规则 4 -4 -5】在 C ++ 中,变量可以“随意申明“,即在哪里使用,就在哪里申明。这很方便。但有时,从人的思维习惯和优化出发,建议在定义后的函数内部,依然像 C 一样,集中申明。

说明　随意申明变量从程序的角度说没有任何问题,但是从人自身的角度来说,会出现以下 3 点问题:

① 若所有的变量都是随意申明的,则当程序比较大时,就有可能出现重复申明的错误。

② 在可执行文件中,数据和代码是分开存放的,如果随意地申明变量,就可能在程序编译中,出现代码和数据混乱地夹杂在一起的情况,这样,程序运行中就可能出现一些意想不到的错误。

③ 随意申明变量不能较好地实现成对编码的思想。因为很多的变量类型(特别是自定义的类),要调用一个释放函数,把类中的一些资源进行释放。如果随意地申明,就很可能忘记应该什么时候进行释放,如果系统中有很多资源没有释放,就可能导致一个程序运行一段时间后,系统就崩溃。

所以,从以上 3 点看出,变量一定要集中申明。

【规则 4-4-6】在分配内存时,如果知道分配的空间较小,建议直接给出一个范围,使用静态分配,而不使用动态分配。

说明　在编程中可能有些程序要使用一块比较小的内存空间,但它的大小一开始就能被确定。

```
int  i;
char  * p;
p = new   char[1024];
for (i = 0;   i < 1023;   i ++)
        {
           p[i] = i;
        }
delete p[ ];
```

这种方法被很多程序员奉为一种经典的用法,他们认为这样做能有效地使用空间,并能提高程序的速度。其实,动态分配一般只有在不能静态分配时才被用到。

动态分配空间有三点不足:

① 动态分配很可能由于某种原因在运行中不能进行内存的分配,这时就可能使整个程序出现错误。如果是静态分配,则系统会在出现资源不足的操作时给出相应的提示。

② 动态分配的空间,不但要自己分配,使用完成后,还要在程序中及时地释放。否则,就可能出现内存漏洞,使操作系统不稳定。但如果是静态分配,分配和释放都是由编译器自己进行管理的,它就能及时地回收,这样就安全多了。

③ 动态分配空间增加了编程量,以上的程序如果用静态分配,只要申明使用就行了。

正例　如果使用的空间本来就比较小,所以只要估计一下程序中要使用的空间的上限就行了。可将以上程序修改为如下的形式:

```
int i
char array[1024];
for(i = 0 ; i < 1023 ; i ++)
        {
           array[i] = i;
        }
```

附录1　C/C++规范检查表

审　查　项	备　注
文件结构	
头文件和定义文件的名称是否合理	
头文件和定义文件的目录结构是否合理	
版权和版本声明是否完整	
头文件是否使用了 ifndef/define/endif 预处理块	*
头文件中是否只存放“声明”而不存放“定义”	
程序的版式	
空行是否得体	
代码行内的空格是否得体	
长行拆分是否得体	
“{”和“}”是否各占一行并且对齐于同一列	
一行代码是否只做一件事。如只定义一个变量则只写一条语句	*
if、for、while、do 等语句自占一行，不论执行语句多少都要加“{}”	*
在定义变量(或参数)时，是否将修饰符 * 和 & 紧靠变量名	*
注释是否清晰并且必要	
注释是否有错误或者可能导致误解	*
类结构的 public、protected、private 顺序是否在所有的程序中保持一致	*
命名规则	
命名规则是否与所采用的操作系统或开发工具的风格保持一致	*
标识符是否直观且可以拼读	
标识符的长度应当符合“min-length && max-information”原则	
程序中是否出现同名的局部变量和全部变量	*
类名、函数名、变量和参数、常量的书写格式是否遵循一定的规则	

续表

审 查 项	备 注
静态变量、全局变量、类的成员变量是否加前缀	
表达式与基本语句	
如果代码行中的运算符较多,是否已经用括号清楚地确定表达式的操作顺序	*
是否编写太复杂或者多用途的复合表达式	
是否将复合表达式与“真正的数学表达式”混淆	*
是否用隐含错误的方式写 if 语句,例如: ① 将布尔变量直接与 TRUE、FALSE 或者 1、0 进行比较 ② 将浮点变量用“==”或“!=”与任何数字比较	*
如果循环体内存在逻辑判断,并且循环次数很大,是否已经将逻辑判断移到循环体的外面	
case 语句的结尾是否忘了加 break	*
是否忘记写 switch 的 default 分支	*
使用 goto 语句时是否留下隐患。例如跳过了某些对象的构造、变量的初始化、重要的计算等	*
常 量	
是否使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串	
在C++程序中,是否用 const 常量取代宏常量	
如果某一常量与其他常量密切相关,是否在定义中包含了这种关系	*
是否误解了类中的 const 数据成员。因为 const 数据成员只在某个对象生存期内是常量,而对于整个类而言却是可变的	
函 数 设 计	
参数的书写是否完整。不要贪图省事只写参数的类型而省略参数名字	
参数命名、顺序是否合理	
参数的个数是否太多	
是否使用类型和数目不确定的参数	
是否省略了函数返回值的类型	
函数名字与返回值类型在语义上是否冲突	

续表

审　查　项	备　注
是否将正常值和错误标志混在一起返回。正常值应当用输出参数获得，而错误标志用return语句返回	*
在函数体的“入口处”，是否用assert对参数的有效性进行检查	*
是否滥用了assert。例如混淆非法情况与错误情况，后者是必然存在的并且是一定要作出处理的	*
return语句是否返回指向“栈内存”的指针或者引用	*
是否使用const提高函数的健壮性。const可以强制保护函数的参数、返回值，甚至函数的定义体，因此应尽可能使用const类型。	
内存管理	
用malloc或new申请内存之后，是否立即检查指针值是否为NULL(防止使用指针值为NULL的内存)	*
是否忘记为数组和动态内存赋初值(防止将未被初始化的内存作为右值使用)	*
数组或指针的下标是否越界	*
动态内存的申请与释放是否配对(防止内存泄漏)	*
是否有效地处理了“内存耗尽”问题	*
是否修改“指向常量的指针”的内容	*
是否出现野指针。例如： ① 指针变量没有被初始化 ② 用free或delete释放了内存之后，忘记将指针设置为NULL	*
是否将malloc/free和new/delete混淆	*
malloc语句是否正确无误。例如字节数是否正确，类型转换是否正确	*
在创建与释放动态对象数组时，new/delete的语句是否正确无误	*
C++函数的高级特性	
重载函数是否有二义性	
是否混淆了成员函数的重载、覆盖与隐藏	*

续表

审 查 项	备 注
运算符的重载是否符合制定的编程规范	
是否滥用内联函数。例如函数体内的代码比较长,函数体内出现循环	
是否用内联函数取代了宏代码	*
类的构造函数、析构函数和赋值函数	
是否违背编程规范而让C++编译器自动为类产生4个缺省的函数: ① 缺省的无参数构造函数 ② 缺省的拷贝构造函数 ③ 缺省的析构函数 ④ 缺省的赋值函数	*
构造函数中是否遗漏了某些初始化工作	*
是否正确地使用构造函数的初始化表	*
析构函数中是否遗漏了某些清除工作	*
是否错写、错用了拷贝构造函数和赋值函数	
赋值函数一般分4个步骤: ① 检查自赋值 ② 释放原有内存资源 ③ 分配新的内存资源,并复制内容 ④ 返回 * this。是否遗漏了重要步骤	*
是否正确地编写了派生类的构造函数、析构函数、赋值函数。注意事项: ① 派生类不可能继承基类的构造函数、析构函数、赋值函数 ② 派生类的构造函数应在其初始化表里调用基类的构造函数 ③ 基类与派生类的析构函数应该为虚(即加 virtual 关键字) ④ 在编写派生类的赋值函数时,注意不要忘记对基类的数据成员重新赋值	*
类的高级特性	
是否违背了继承和组合的规则 ① 若在逻辑上B是A的“一种”,并且A的所有功能和属性对B而言都有意义,则允许B继承A的功能和属性 ② 若在逻辑上A是B的“一部分”(a part of),则不允许B从A派生,而是要用A和其他东西组合出B	*

续表

审　查　项	备　注
其他常见问题	
数据类型问题： ① 变量的数据类型有错误吗 ② 存在不同数据类型的赋值吗 ③ 存在不同数据类型的比较吗	*
变量值问题： ① 变量的初始化或缺省值有错误吗 ② 变量发生上溢或下溢吗 ③ 变量的精度够吗	*
逻辑判断问题： ① 由于精度原因导致比较无效吗 ② 表达式中的优先级有误吗 ③ 逻辑判断结果颠倒了吗	*
循环问题： ① 循环终止条件不正确吗 ② 无法正常终止(死循环)吗 ③ 错误地修改循环变量了吗 ④ 存在误差累积吗	*
错误处理问题： ① 忘记进行错误处理了吗 ② 错误处理程序块一直没有机会被运行吗 ③ 错误处理程序块本身就有毛病吗。如报告的错误与实际错误不一致，处理方式不正确等 ④ 错误处理程序块是“马后炮”吗。如它在被调用之前软件已经出错	*
文件 I/O 问题： ① 对不存在的或者错误的文件进行操作了吗 ② 文件以不正确的方式打开了吗 ③ 文件结束判断不正确吗 ④ 没有正确地关闭文件吗	*

注：“*”表示比较重要的项目。

附录 2　Delphi 标准控件的命名参考

附表 2－1　Standard 页控件

类　　名	命 名 格 式
TMainMenu	mmnuXXXX
TPopupMenu	pmnuXXXX
TLabel	lblXXXX
TEdit	edtXXXX
TMemo	mmoXXXX
TButton	btnXXXX
TCheckBox	chkXXXX
TRadioButton	rdbtXXXX
TListBox	lbxXXXX
TComboBox	cbxXXXX
TScrollBar	scbrXXXX
TGroupBox	gbxXXXX
TRadioGroup	rgpXXXX
TPanel	pnlXXXX
TActionList	actsXXXX

附表 2－2　Additional 页控件

类　　名	命 名 格 式
TBitBtn	btbtXXXX
TSpeedButton	spbtXXXX
TMaskEdit	medtXXXX
TStringGrid	sgrdXXXX
TDrawGrid	dgrdXXXX
TImage	imgsXXXX
TShape	shpXXXX
TBevel	bvlXXXX
TScrollBox	scbxXXXX
TCheckListBox	chksXXXX
TSplitter	spltXXXX
TStaticText	stxtXXXX
TControlBar	ctbrXXXX
TChart	chtXXXX

附表 2－3　Win32 页控件

类　　名	命 名 格 式
TTabControl	tctlXXXX
TPageControl	pctlXXXX
TImageList	imgsXXXX
TRichEdit	redtXXXX
TTrackBar	tkbrXXXX
TProgressBar	pgbrXXXX
TUpDown	updnXXXX
THotKey	htkXXXX
TAnimate	anmtXXXX
TDateTimePicker	dtpXXXX
TMonthCalendar	mcldXXXX
TTreeView	tvwXXXX
TListView	lvwXXXX
THeaderControl	hctlXXXX
TStatusBar	stbrXXXX
TToolBar	tlbrXXXX
TCoolBar	clbrXXXX
TPageScroller	psclXXXX

附表 2－4　System 页控件

类　　名	命 名 格 式
TTimer	tmrXXXX
TPaintBox	ptbxXXXX
TMediaPlayer	mplrXXXX
TOLEContainer	olectXXXX
TDDEClientConv	ddeccXXXX
TDDEClientItem	ddeciXXXX
TDDEServerConv	ddescXXXX
TDDEServerItem	ddesiXXXX

附表 2－5　Internet 页控件

类　名	命 名 格 式
TClientSocket	XXXXClientSocket
TServerSocket	XXXXServerSocket
TWebDispatcher	XXXXWebDispatcher
TPageProducer	XXXXPageProducer
TQueryTableProducer	XXXXQTProducer
TDataSetTableProceducer	XXXXDSTProducer
TDataSetPageProceducer	XXXXDSPProducer
TNMDayTime	XXXXNMDayTime
TNMEcho	XXXXNMEcho
TNMFinger	XXXXNMFinger
TNMFTP	XXXXNMFTP
TNMHTTP	XXXXNMHTTP
TNMMsg	XXXXNMMsg
TNMMSGServ	XXXXNMMsgServ
TNMNNTP	XXXXNMNNTP
TNMPOP3	XXXXNMPOP3
TNMUUProcessor	XXXXNMUUProcessor
TNMSMTP	XXXXNMSMTP
TNMStrm	XXXXNMStrm
TNMStrmServ	XXXXNMStrmServ
TNMTime	XXXXNMTime
TNMUDP	XXXXNMUDP
TPowerSock	XXXXPowerSock
TNMGeneralServer	XXXXNMGeneralServer
THTML	XXXXHTML
TNMURL	XXXXNMURL

附表 2-6　Data Access 页控件

类　名	命 名 格 式
TDataSource	dsXXXX
TTable	tblXXXX
TQuery	qryXXXX
TStoredProc	sprcXXXX
TDatabase	dbXXXX
TSession	ssnXXXX
TBatchMove	bmvXXXX
TUpdateSQL	updtXXXX
TNestedTable	ntblXXXX

附表 2-7　Data Controls 页控件

类　名	命 名 格 式
TDBGrid	dbgrdXXXX
TDBNavegator	dbnvXXXX
TDBText	dbtxtXXXX
TDBEdit	dbedtXXXX
TDBMemo	dbmmoXXXX
TDBImage	dbimgXXXX
TDBListBox	dblbxXXXX
TDBComboBox	dbcbxXXXX
TDBCheckBox	dbchkXXXX
TDBRadioGroup	dbrgpXXXX
TDBLookupListBox	dbllbXXXX
TDBLookupComboBox	dblcbXXXX
TDBRichEdit	dbredXXXX
TDBCtrlGrid	dbcgdXXXX
TDBChart	dbchtXXXX

附表 2－8　Midas 页控件

类　名	命名格式
TClientDataSet	cltdsXXXX
TDCOMConnection	dcmcnXXXX
TCorbaConnection	crbcnXXXX
TSocketConnection	sktcnXXXX
TOLEnterpriseConnection	olecnXXXX
TDataSetProvider	dsprdXXXX
TProvider	prvdrXXXX
TSimpleObjectBroker	sobkrXXXX
TRemoteServer	rmtsvXXXX
TMidasConnection	mdscnXXXX

附表 2－9　Decision Cube 页控件

类　名	命名格式
TDecisionCube	dccbXXXX
TDecisionQuery	dcqryXXXX
TDecisionSource	dcsrXXXX
TDecisionPivot	dcpvtXXXX
TDecisionGrid	dcgrdXXXX
TDecisionGraph	dcchtXXXX

附表 2－10　QReport 页控件

类　名	命名格式
TQuickRep	qrptXXXX
TQRSubDetail	qrsdtXXXX
TQRStringsBand	qrsbdXXXX
TQRBand	qrbdXXXX
TQRChildBand	qrcbdXXXX
TQRGroup	qrgrpXXXX
TQRLabel	qrlblXXXX
TQRDBText	qrdbtXXXX
TQRExpr	qrxprXXXX

续表

类　名	命名格式
TQRSysData	qrsysXXXX
TQRMemo	qrmmoXXXX
TQRExprMemo	qrxpmXXXX
TQRRichEdit	qrredXXXX
TQRDBRichEdit	qrdbrXXXX
TQRShape	qrshpXXXX
TQRImage	qrimgXXXX
TQRDBImage	qrdbiXXXX
TQRCompositeReport	qrcrpXXXX
TQRPreview	qrprvXXXX
TQRTextFilter	qrtflXXXX
TQRCSVFilter	qrcsvXXXX
TQRHTMLFilter	qrhtfXXXX
TQRChart	qrchtXXXX

附表 2－11　Dialogs 页控件

类　名	命名格式
TOpenDialog	opdgXXXX
TSaveDialog	svdgXXXX
TOpenPictureDialog	oppdgXXXX
TSavePictureDialog	svpdgXXXX
TFontDialog	ftdgXXXX
TColorDialog	cldgXXXX
TPrintDialog	prdgXXXX
TPrinterSetupDialog	psdgXXXX
TFindDialog	fndgXXXX
TReplaceDialog	rpdgXXXX

附表 2－12　Win 3.1 页控件

类　　名	命 名 格 式
TDBLookupList	dbllsXXXX
TDBLookupCombo	dblcbXXXX
TTabSet	tbstXXXX
TOutLine	otlnXXXX
TTabbedNoteBook	tnbkXXXX
TNoteBook	ntbkXXXX
THeader	hdrXXXX
TFileListBox	flbxXXXX
TDirectoryListBox	dlbxXXXX
TDriveComboBox	dcbxXXXX
TFilterComboBox	fcbxXXXX

附表 2－13　Samples 页控件

类　　名	命 名 格 式
TGauge	ggeXXXX
TColorGrid	clgrdXXXX
TSpinButton	spnbtXXXX
TSpinEdit	spedtXXXX
TDirectoryOutline	dirolXXXX
TCalendar	cldrXXXX
TIBEventAlerter	ibaltXXXX

附表 2－14　ActiveX 页控件

类　　名	命 名 格 式
TChartFX	chtfxXXXX
TVSSpell	vssplXXXX
TF1Book	bkf1XXXX
TVtChart	vtchtXXXX
TGraph	grphXXXX

注：1. 可只对进行编码的控件命名，如果某控件只用作显示，如 TLabel，则可采用 Delphi 自动生成的名字 Lable1。

2. 若使用第三方控件或自定义类型控件，命名规则应以以上规则为参考。

附录3　VC++优化编码实现过程示例

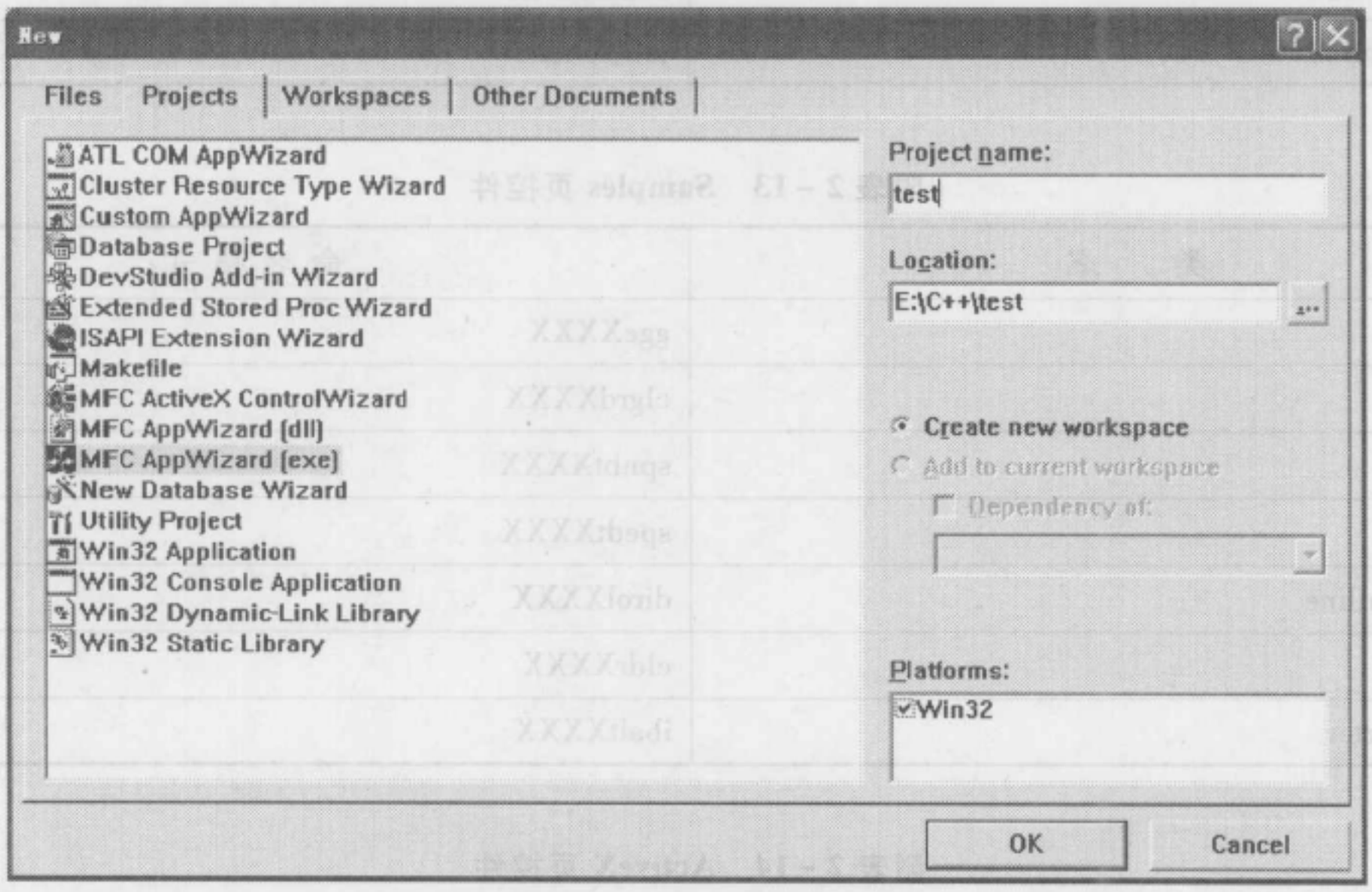
下面,通过一个实例来说明优化编码等方面的规则。

1. 系统设置

(1) 建立工程

首先,按如下步骤建立一个演示工程。

① 在VC的向导中生成一个基本的对话框的工程,如图附图3.1所示。输入工程的名称“test”,单击“OK”按钮。



附图3.1　利用向导生成一个基本对话框的工程

② 选取应用程序的类型为“Dialog based”,单击“Finish”按钮,如图附图3.2所示。

③ 建立演示工程完成。

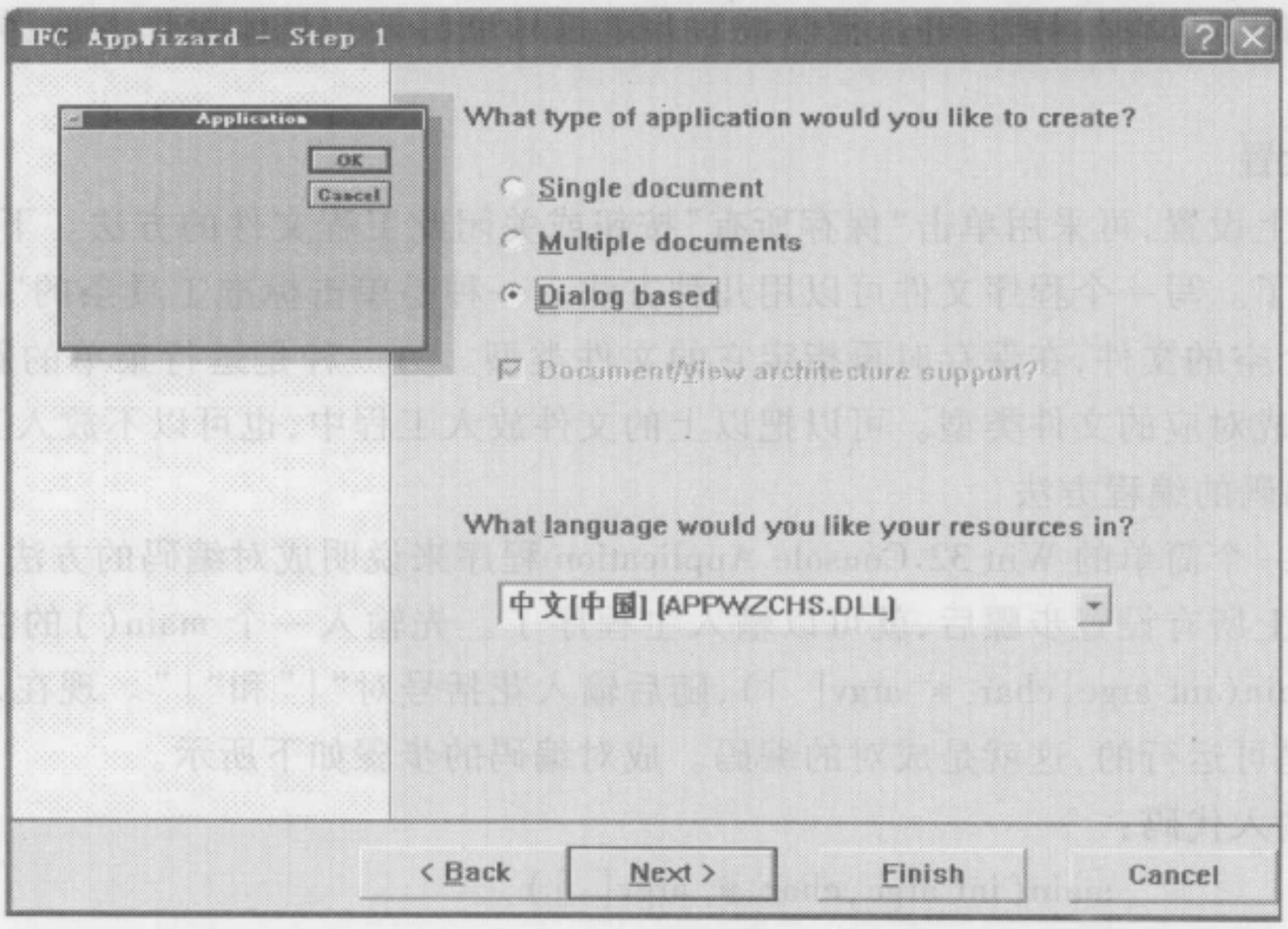
(2) 编写代码前的工作

① 去掉PCH垃圾。

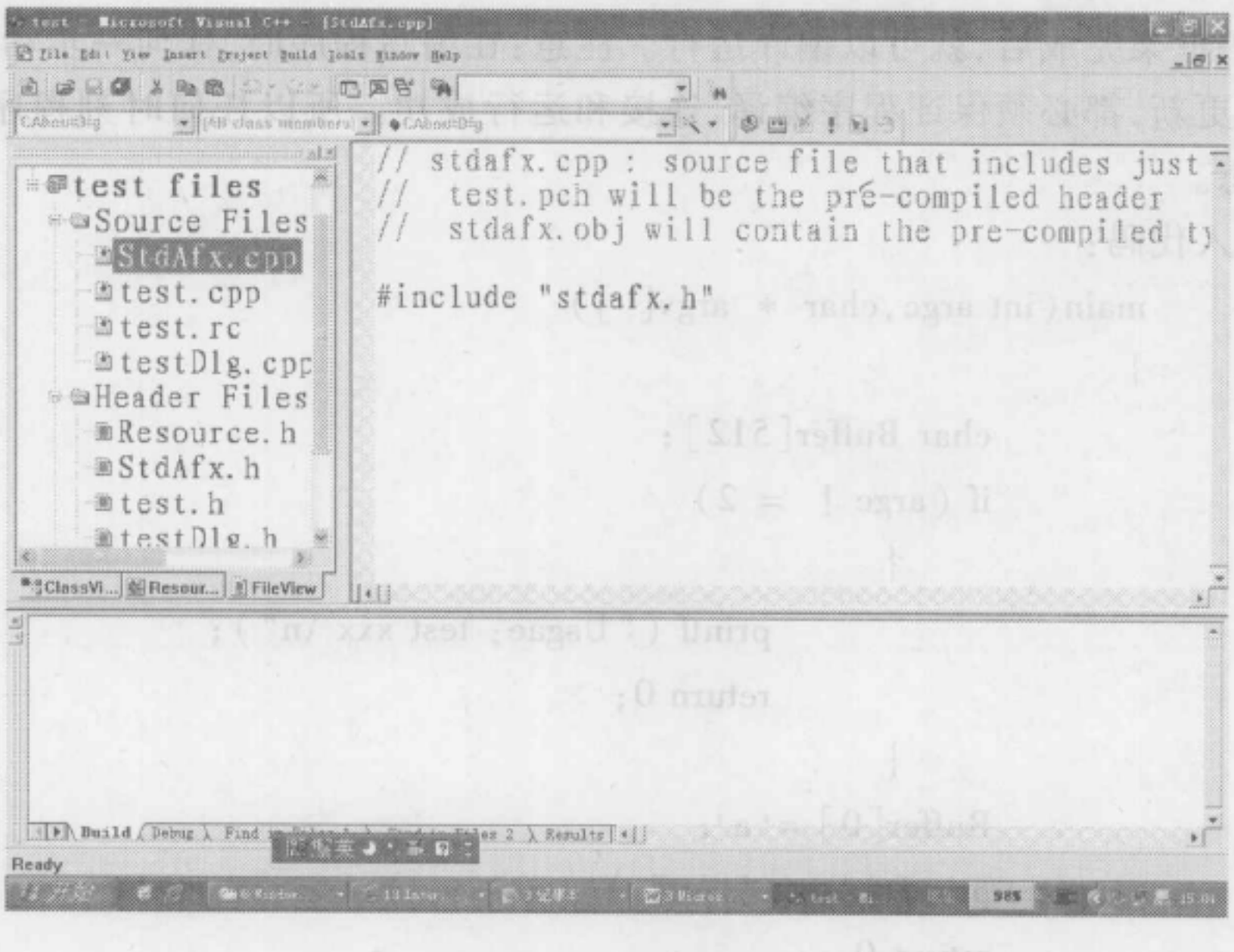
打开在“Workspace”的“FileView”选项卡,从中选取“StdAfx.cpp”,单击“Delete”键删除这个文件,工程的界面如附图3.3所示。

可以看到,在StdAfx.cpp的文件中根本没有什么内容,只有一句#include"stdafx.h"。这就是典型的垃圾,应删除该文件,详见第四部分第1章相关内容。

② 在Release模式下写代码。



附图 3.2　应用程序类型为“Dialog based”



附图 3.3　工程界面图

很多人喜欢在 Debug 模式下编写程序，当完成代码编写后，再去生成 Release 来使用。这样，就经常出现在 Debug 模式下编写出的程序不能在 Release 模式中运行的情况。这种情况在 MFC 编写的程序中经常出现。要避免这种情况，最好一开始就在 Release 的模式中工作，而不是在 Debug 模式中工作。

但在标准的 Release 模式下，是没有调试信息的，那怎么办呢？其实很简单，即在 Release 模

式中把源代码的调试符号打开,并且把链接的调试符号也打开。具体操作方法详见第四部分第1章相关内容。

③ 存储设置

要保存以上设置,可采用单击“保存所有”按钮或关闭此工程文件的方法。下面可以开始写工程中的程序了。写一个程序文件可以用几种方法。一种是单击标准工具条的“New Text File”按钮,生成一个空的文件,在保存时再指定它的文件类型。另一种是运行菜单的新建项,在弹出的对话框中生成对应的文件类型。可以把以上的文件放入工程中,也可以不放入工程中。

2. 成对编码的编程方法

下面通过一个简单的 Win 32 Console Application 程序来说明成对编码的方法。

在完成以上所有设置步骤后,就可以输入主程序了。先输入一个 main( )的主函数,然后输入它的参数 main(int argc,char * argv[ ]),随后输入花括号对“{”和“}”。现在,程序函数框架就完成了,它是可运行的,这就是成对的编码。成对编码的步骤如下所示。

第一步,输入代码:

```
main(int argc,char * argv[ ])
{
}
```

当一个代码框架完成后,就可以编译运行。注意:在编写程序时,任何一步动作,无论是写新代码,还是修改更新,都必须保证程序编译、连接和运行成功。所以任何时刻程序总是保持在一种可运行的状态。

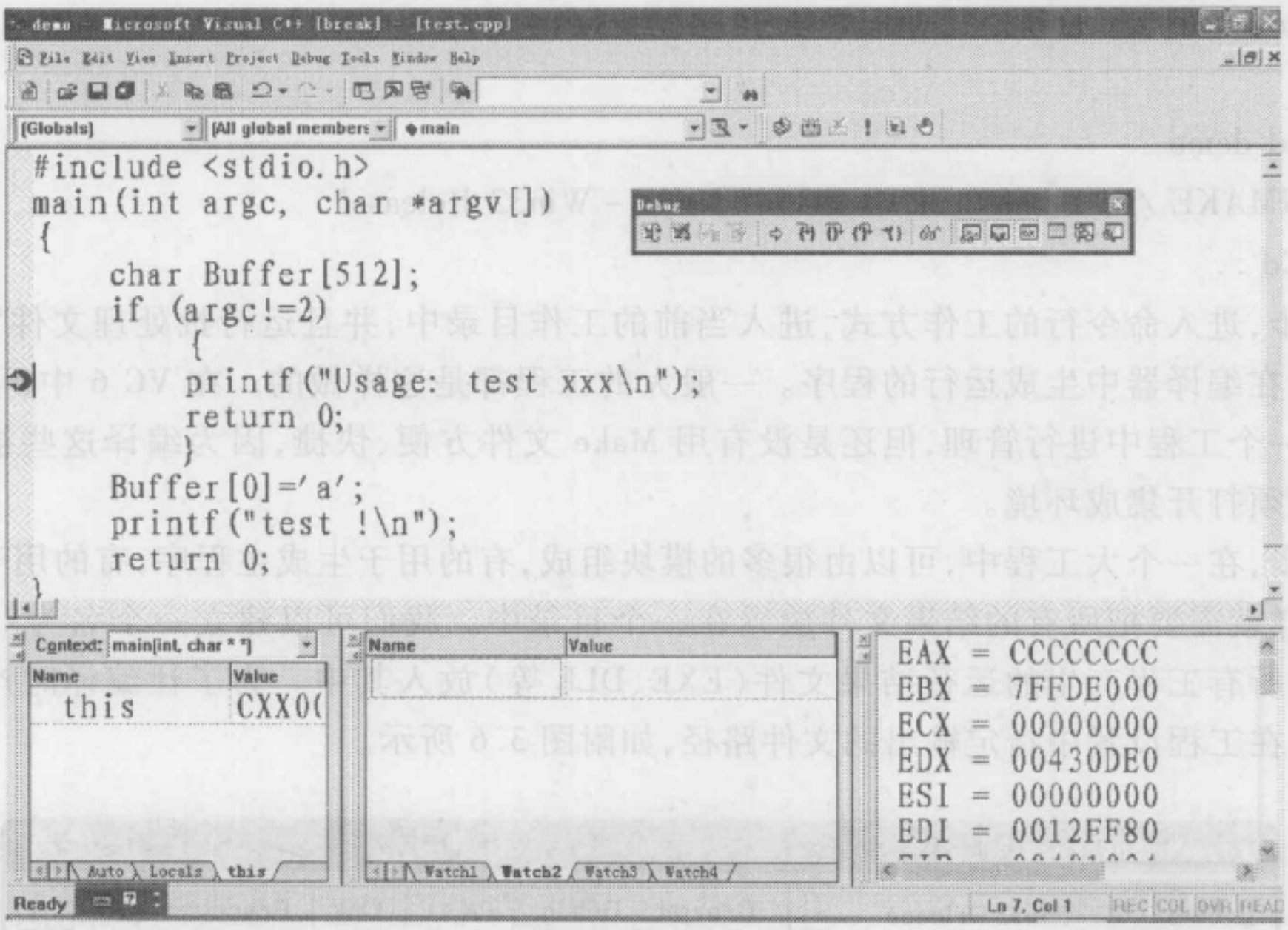
第二步,输入代码:

```
main(int argc,char * argv[ ])
{
    char Buffer[512];
    if (argc ! = 2)
        {
            printf ("Usgae: test xxx \n");
            return 0;
        }
    Buffer[0] ='a';
    printf ("test ! \n" ) ;
    return 0;
}
```

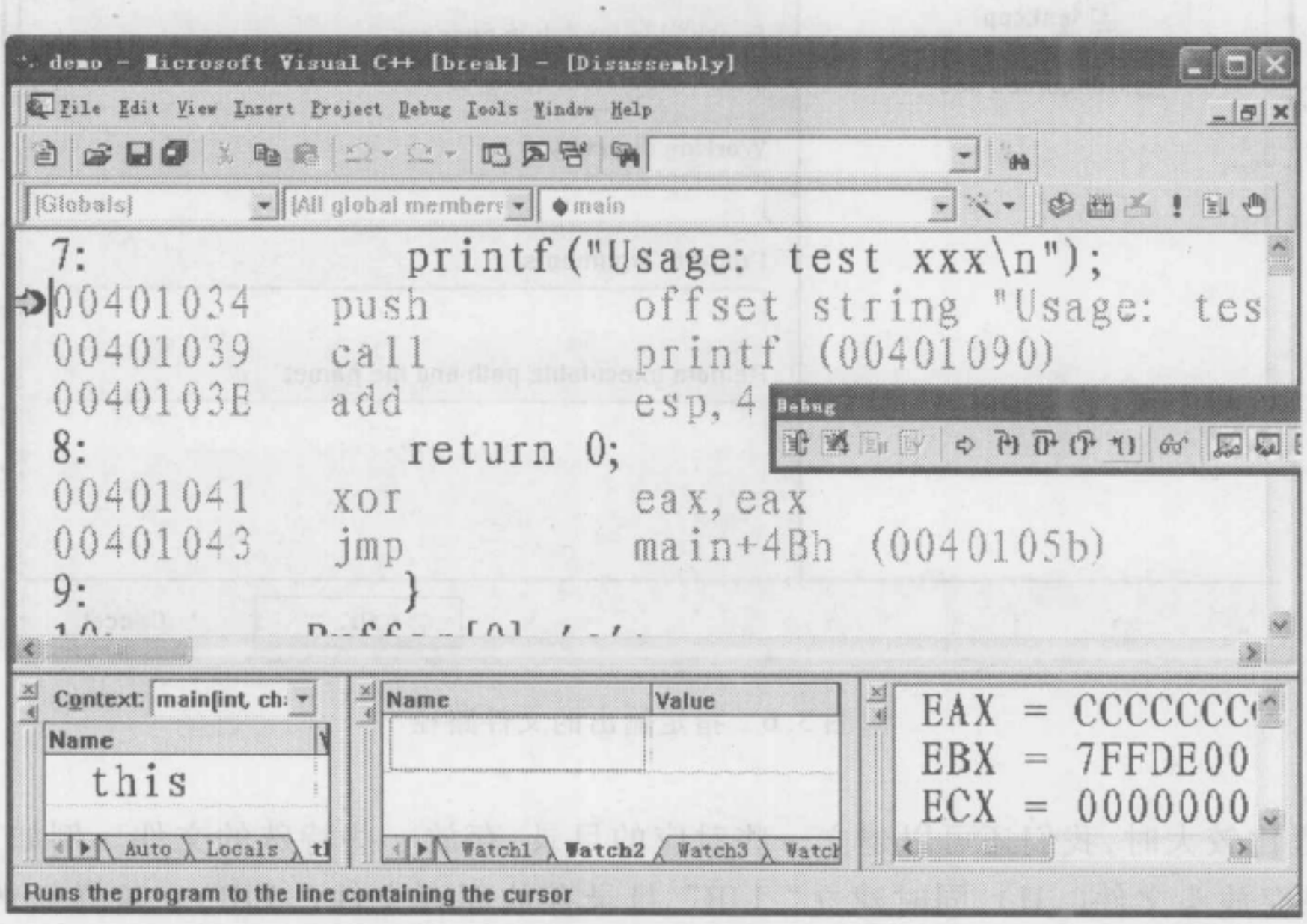
第三步,编译程序运行并可以开始调试。在调试前,可以先设置一个断点,然后单击 F5 键,开始进入调试界面。如附图 3.4 所示。

第四步,打开调试工具条的“ Disassembles”项,就可以看到 C 语言对应的汇编代码(一般去掉调试说明信息),如附图 3.5 所示。

第五步,当整个工程编写完成后,需创建一个名为“Build”的批处理文件。在 VC 4 中,本身就是用 Make 文件作为工程的文件,但在 VC 6 中,把 Make 文件改成了 DSP 文件了。



附图3.4　调试窗口



附图3.5　C语言对应的汇编代码

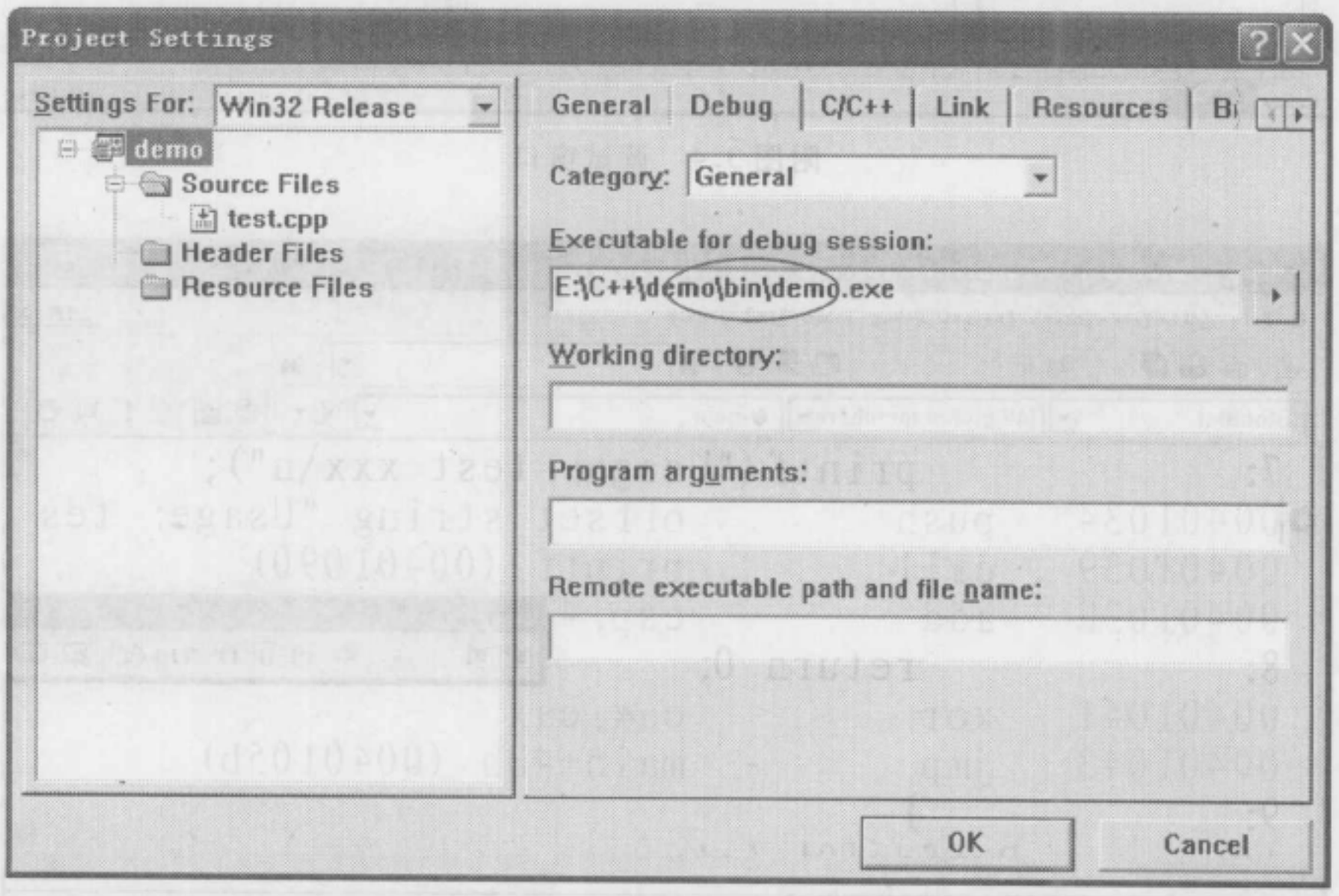
为了创建一个 Make 文件，可以执行“Project”菜单下的“Export Makefile ...”菜单项。选取要输出的文件，单击“OK”按钮，Make 文件就会被生成。接下来创建一个“ Build”的处理文件，在文件中输入如下的文本。

```
call vc6path. bat

cd demo
NMAKE /f "Demo. mak"  CFG = "demo  - Win32 Release"
Cd . .
```

第六步,进入命令行的工作方式,进入当前的工作目录中,并且运行批处理文件“Build”,此时,就可能在编译器中生成运行的程序。一般大的工程都是这样做的。在 VC 6 中,可以把多个工程放入一个工程中进行管理,但还是没有用 Make 文件方便、快捷,因为编译这些被集成的工程时,都必须打开集成环境。

第七步,在一个大工程中,可以由很多的模块组成,有的用于生成主程序,有的用于生成 DLL 文件。这样就需要把所有的结果文件放置在一个目录中。我们可以建立一个名为“ BIN”的文件目录,把所有工程产生的运行结果文件(EXE、DLL 等)放入其中。为了让编译时产生统一的输出,可以在工程设置中指定输出的文件路径,如附图 3.6 所示。



附图 3.6　指定输出的文件路径

当工程比较大时,我们还可以建立一些对应的目录,存放一些特殊的文件。例如,建立“ Include”目录存放头文件(.H),同时建立“ LIB” 目录存放库存文件(.LIB)。这样做的目的就是方便管理。程序运行时也能找到所需的资源,VC 本身的集成环境就是这样做的。

最后,删除那些在集成环境中不必要的文件和目录。例如,DEP 文件是必须删除的文件,否则这个工程将不仅不能改变目录,还可能使得用 build 文件编译时不成功。还可以进一步删除某些集成环境的文件,例如 DSW、NCB、OPT 文件等。

参考文献

1 谭浩强. C++程序设计. 北京:清华大学出版社,2004

2 陈世忠. C++编码规范. 北京:人民邮电出版社,2002

3 Lippman S B, Lajoie J. C++ Primer(3RD)中文版. 潘爱民,张丽译. 北京:中国电力出版社,2005

4 中国标准出版社. 计算机软件工程规范国家标准汇编. 北京:中国标准出版社,1998

5 梁肇新. 编程高手箴言. 北京:电子工业出版社,2003

6 Gosling J, Joy B, Steele G. Java 语言规范. 蒋国新,朱暄,李洪伟译. 北京:北京大学出版社,1997

7 Lindholm T, Yellin F. Java 虚拟机规范. 玄伟剑,陈永智,蔡伟译. 北京:北京大学出版社,1997

8 Bronson G J. Java 编程原理:面向工程和科学人员. 张珑,刘雅文译. 北京:清华大学出版社,2004

9 聂哲,袁梅冷,杨淑萍. Java 面向对象程序设计. 北京:高等教育出版社,2005

10 Teixeira S, Pacheco X. Delphi 6 开发人员指南. 龙劲松译. 北京:机械工业出版社,2003

11 Schildt H, Guntle G. C++ Builder 技术大全. 周海斌译. 北京:机械工业出版社,2002

12 李俊平,薛海燕. Delphi 面向对象程序设计. 北京:高等教育出版社,2005

13 刘艺. Delphi 6 企业级解决方案及应用剖析. 北京:机械工业出版社,2002

14 Foxall J. Visual Basic. NET 编程标准. 附昭伟译. 北京:清华大学出版社,2003

15 林锐,顾晓刚,谢义军. 高质量程序设计指南. 北京:电子工业出版社,2002

16 Humphrey W S. 软件工程规范. 傅为译. 北京:清华大学出版社,2004

17 余苏宁,王明福. C++程序设计. 北京:高等教育出版社,2004

18 王明福,余苏宁. Visual C++程序设计. 北京:高等教育出版社,2004

19 Deitel H M, Deitel P J. C++大学教程. 第2版. 邱仲潘译. 北京:电子工业出版社,2001

20 Stevens A. C++大学自学教程. 林瑶译. 北京:电子工业出版社,2004